NTNU – Trondheim
Norwegian University of
Science and Technology

Mechanism parametrization, modeling and FE-meshing

Rasmus Korvald Skaare

Mechanical Engineering
Submission date: June 2015
Supervisor: Ole Ivar Sivertsen, IPM

Norwegian University of Science and Technology
Department of Engineering Design and Materials

THE NORWEGIAN UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF ENGINEERING DESIGN
AND MATERIALS

MASTER THESIS SPRING 2015
FOR
STUD.TECHN. RASMUS KORVALD SKAARE

Mechanism Parametrization, Modeling and FE meshing

Parametrisering, modellering og FE-meshing av mekanismer

Knowledge Based Engineering (KBE) is a technology that is typically used for introducing automation in engineering work. This technology has been used with great success in Aker Solutions over the last 10-15 years based on a technology developed by the company TechnoSoft Inc. in Ohio, USA. The KBE applications are based on an object oriented programming languages called AML supported by the TechnoSoft Company. This master assignment is based on a pilot implementation in AML for mechanism modeling that was tested and developed in the project assignment last autumn. During the project assignment the candidate also studied the Sheth-Uicker (SU) convention and the master assignment will also be used to study mechanism modeling based on the SU convention.

The master assignment includes the following:

1. Litterateur study of applications of the SU conventions in mechanism design
2. Litterateur study regarding FE mesh generation.
3. Modeling mechanism links based on the SU convention
4. Compare the link modeling techniques used in the project assignment with modeling techniques based on the SU convention and discuss possible conversions between the two
5. Implement mesh generation for the mechanism links using functionality in AML including joint connection points
6. If time allow, implement an integrated mechanism modeling pilot for automated generating of FE based simulation input and possibly use of optimization techniques

Formal requirements:

Three weeks after start of the thesis work, an A3 sheet illustrating the work is to be handed in. A template for this presentation is available on the IPM's web site under the menu "Masteroppgave" (http://www.ntnu.no/ipm/masteroppgave). This sheet should be updated one week before the master's thesis is submitted.

Risk assessment of experimental activities shall always be performed. Experimental work defined in the problem description shall be planed and risk assessed up-front and within 3 weeks after receiving the problem text. Any specific experimental activities which are not properly covered by the general risk assessment shall be particularly assessed before performing the experimental work. Risk assessments should be signed by the supervisor and copies shall be included in the appendix of the thesis.

The thesis should include the signed problem text, and be written as a research report with summary both in English and Norwegian, conclusion, literature references, table of contents, etc. During preparation of the text, the candidate should make efforts to create a well arranged and well written report. To ease the evaluation of the thesis, it is important to cross-reference text, tables and figures. For evaluation of the work a thorough discussion of results is appreciated.

The thesis shall be submitted electronically via DAIM, NTNU's system for Digital Archiving and Submission of Master's theses.

The contact person:
Bjørn Haugen, NTNU, Department of Engineering Design and Materials.

[illegible]

Torgeir Welo
Head of Division

Ole Ivar Sivertsen

Ole Ivar Sivertsen
Professor/Supervisor

NTNU
Norges teknisk-naturvitenskapelige universitet
Institutt for produktutvikling og materialer

Preface

This is a Master's Thesis written in the course TMM4901 - Engineering Design, Calculation and Manufacture, accomplished during the spring 2015 at the Department of Engineering Design and Materials, Norwegian University of Science and Technology (NTNU) in Trondheim.

The thesis is a continuation of my project thesis fall 2014 based on a KBE pilot implementation in AML developed by Ole Ivar Sivertsen during a sabbatical year (2013-2014) when he was visiting TechnoSoft Inc. in Ohio, USA.

I would like to thank Professor Ole Ivar Sivertsen for his never-ending enthusiasm, guidance, feedback and lots of inspiring discussions throughout the entire duration of the thesis work. Much appreciated help was also given by Professor Bjørn Haugen, as well as Trapper Schuler and TechnoSoft Inc. providing the AML software used and technical support.

Trondheim, June 17, 2015

Rasmus Korvald Skaare

Abstract

This thesis is written to show the development of an application used for mechanism parametrization, modeling and finite element meshing. Concepts from Knowledge Based Engineering (KBE) are applied and the possibility of using the Sheth – Uicker convention (SU) for parameterizing the links and joints of a mechanism is explored.

A KBE application has been developed in the Adaptive Modeling Language (AML) implementing design rules and mechanism theory in a parametric model. The application features customization of the shape, dimension and cross section of links regardless of the number of joints connected. Incidence relationships together with joint position, type and direction are given as input and dimensions are calculated based on the connectivity of the mechanical system. The application also automatically meshes the model.

From the work done automated modelling and meshing of mechanisms is demonstrated from given input parameters, and is an important step on the way to automate the entire mechanism design process. The application provides concept and ideas and a solid framework for further development that has the potential to drastically reduce routine work in the design process of mechanisms.

Sammendrag

Denne rapporten er skrevet for å vise utviklingen av en applikasjon brukt for parametrisering, modellering og finite element meshing av mekanismer. Konsepter fra Kunnskapsbasert Engineering (KBE) er tatt i bruk og muligheten for å bruke Sheth og Uickers konvensjon (SU) for parametrisering av lenker og ledd i en mekanisme er studert.

En KBE applikasjon har blitt utviklet i Adaptive Modeling Language (AML) ved å implementere designregler og mekanismeteori i en parametrisk modell. Applikasjonen inneholder funksjonalitet for tilpasning av form, dimensjon og valg av type tverrsnitt på lenker uavhengig av antallet av ledd den er tilkoblet. Koblingene sammen med leddens posisjon, type og retning er gitt som inndata og dimensjonene er beregnet basert på koblingene av mekansimen. Applikasjonen mesher også modellen automatisk.

Ut fra arbeidet som er gjennomført blir automatisert modellering og meshing av mekanismer demonstrert fra gitte inndataparametre, og dette er et viktig skritt på veien for å automatisere hele designprosessen av mekanismer. Applikasjonen presenterer konsepter og ideer og er et solid rammeverk for videre utvikling som har potensial til å drastisk redusere rutinearbeid i designprosessen av mekanismer.

Table of Contents

List of Figures

Notations

General notation

A vector is denoted by a small bold letter, e.g., $\boldsymbol{a}$, a matrix by capital bold letter, e.g., $\boldsymbol{M}$. Lines, links, joints, and frames are denoted by capital letters. For operations, the following are needed: the sign · indicates a scalar, vector, or matrix multiplication. In this thesis, the notation of transposing $(.\ )^T$ is not used for vectors: instead of writing $\boldsymbol{a}^T \cdot \boldsymbol{b}$, a shorter notation is used with the symbol $*$ to indicate the sum of element-wise multiplications, such that $\boldsymbol{a} * \boldsymbol{b} = \sum a_i \cdot b_i$. Two orthogonal projections are used in this thesis. First, the orthogonal projection of a vector $\boldsymbol{b} \in \mathbb{R}^d$ onto some vector $\boldsymbol{a} \in \mathbb{R}^d$ is denoted as $\pi_{\mathbf{a}}(\boldsymbol{b})$.

$$\pi_{\mathbf{a}}(\boldsymbol{b}) = \frac{\boldsymbol{b} * \boldsymbol{a}}{\boldsymbol{a} * \boldsymbol{a}} \cdot \boldsymbol{a} = \underset{\kappa \cdot \boldsymbol{a}, \kappa \in \mathbb{R}}{\operatorname{argmin}}\, dist(\kappa \cdot \boldsymbol{a}, \boldsymbol{b})$$

Second, the orthogonal projection of a vector $\boldsymbol{b} \in \mathbb{R}^d$ into the orthogonal complement $\boldsymbol{a}^\perp \in \mathbb{R}^d$ of some vector $\boldsymbol{a}$ is denoted as $\tau_{\mathbf{a}}(\boldsymbol{b})$.

$$\tau_{\mathbf{a}}(\boldsymbol{b}) = \boldsymbol{b} - \frac{\boldsymbol{b} * \boldsymbol{a}}{\boldsymbol{a} * \boldsymbol{a}} \cdot \boldsymbol{a} = \underset{\boldsymbol{p} \in \boldsymbol{a}^\perp \in \mathbb{R}}{\operatorname{argmin}}\, dist(\boldsymbol{p}, \boldsymbol{b})$$

Latin lower case letters

$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$	axes-vectors of frame F
$\boldsymbol{p}$	location vector
$\boldsymbol{\omega}$	direction vector
$\boldsymbol{v}_0$	orthogonal moment vector

Latin upper case letters

F	frame, local coordinate system

$F_{(ij)_i}, F_{(ij)_j}$	frames at joint J_{ij}
G_{ij}, G_{jk}	lines
J_{ij}	joint connecting L_i and L_j
L_i	link
Z_{ij}, Z_{jk}	lines through joint axes on J_{ij} and J_{jk}

Abbreviations

AI	Artificial Intelligence
AML	Adaptive Modeling Language
CAD	Computer Aided Design
CAE	Computer Aided Engineering
DOF	Degree Of Freedom
FEA	Finite Element Analysis
GUI	Graphical User Interface
KBE	Knowledge Based Engineering
MPC	Multi-Point Constraint
NURBS	Non-uniform rational basis spline
OOP	Object Oriented Programming
PDE	Partial Differential Equation
REPL	Read Eval Print Loop
SU	Sheth – Uicker
TSI	TechnoSoft Inc.

1 Introduction

1.1 Background

Automation in engineering design is typically referred to as Knowledge Based Engineering (KBE) (Rocca, 2012), (Pinfold & Chapman, 1999). Over the last 15 years, KBE has been successful in a wide variety of defense and commercial applications including aerospace, automotive, and capital equipment, resulting in technologies like KBeDesign™ from Aker Solutions[1] and Collaborative Hypersonic Airbreathing Vehicle Environment (CoHAVE) at NASA (Chemaly, 2006). KBE has for many years been the exclusive domain of a few and highly competitive industries (aerospace and automotive in particular) and has not entered mainstream academic research.

A mechanical system, or a mechanism, is defined as a collection of links which may be interconnected by joints that constrain the relative motions between them. In the design process of a mechanical system there are three basic stages; type synthesis (selecting the type of mechanism), number synthesis (deciding the numbers of links and the types and numbers of joints connecting them) and dimensional synthesis (assigning detailed shapes, dimensions and material properties). (Uicker, Ravani, & Sheth, 2013)

Modelling and simulating a mechanical system using CAE is extensive work, requiring each link and joint to be modelled, assembled and meshed, before running the simulation and analyzing the results. If this requires modifications of any shapes or dimensions it will most certainly cause discrepancies leading to further remodeling. Changing any decisions made earlier in the design process, like the type of mechanism or number and type joints or links, will require most of subsequent work to be redone. This process is repeated several times before the final design is ready for production. (Sivertsen, 2014)

Introducing automation in the design process of mechanisms will reduce routine design and repetitive work. Using a KBE system with integrated design and simulation tools can implement design rules and mechanism theory in a parametric model. A geometric model is then generated and idealized, and meshing and simulation are run automatically. An optimization is then run comparing the design

[1] http://www.kbedesign.com

against performance criteria, generating new input to the model and iterating towards the final design.

To create a parametric model a representation for mechanisms are needed. It should hold the essential parameters, describing the "shape" and motion of each link, for a complete description of the mechanism. Bongardt (2013) reviews different conventions for specifying kinematics of mechanisms and emphasizes the practical and theoretical preferences of the generalized convention developed by Sheth and Uicker (1971). This convention delivers sets of parameters used to parametrize joint- and link displacements. This representation enables comparison between the topology and the geometry of different mechanisms.

1.2 Research questions

The research questions defined below will give the background for the work done in this report.

RQ1: How can Sheth – Uicker convention be utilized to automate the generation of link geometry?

RQ2: How can different links be represented using one generic class?

RQ3: How will the physical dimensions of links and joints affect each other?

RQ4: How can mesh generation be automated to create a link between modeling and simulation?

1.3 Related work

The basis and inspiration for the work conducted in this thesis is based on the *Proposed Approach for Introducing Automation in Mechanism Design* by Sivertsen (2014). Sivertsen (2014) presents key concepts and elements for implementing Knowledge Based Engineering (KBE) in automation of the design process for mechanisms and developed a pilot implementation using the TechnoSoft Inc. (TSI) supported Adaptive Modeling Language (AML). This is a mechanism analysis and simulation model based on a Finite Element (FE) algorithms. The application parameterizes the joint positions and from that automatically generates the links and joints. The application also features a way to make a solid model of the mechanism. Classes for different types of common mechanisms are predefined, and default shapes for links and joints are provided.

1.4 Structure

Chapter 2 presents theory and research from KBE, the SU convention and mesh generation. Chapter 3 is a review of the tools and methods used for the work with this thesis. Chapter 4 presents the development of the mechanism application and looks into modeling links based on the SU convention and mesh generation in AML. The results are presented in chapter 5 and in chapter 6 the research questions are discussed. Conclusions are presented in chapter 7 and suggestions for further work are found in chapter 8.

2 Theory

2.1 Knowledge Based Engineering

Knowledge Based Engineering (KBE) is the art of using computerized knowledge to automate engineering design. Software tools used to practice KBE offer an advanced modeling paradigm with an open architecture, enabling the automation of the entire product development cycle, integrating product configuration, design, analysis, visualization, production planning, inspection, and cost estimation.

Pinfold and Chapman (2001) called it "*an engineering method that represents a merging of object oriented programming (OOP), artificial intelligence(AI) techniques and computer-aided design technologies, giving benefit to customized or variant design automation solutions*".

Developing a KBE application is mostly about writing code using a KBE programming language. But before starting with development, the typical methodological approach to practice KBE is to start with the identification, then the acquisition and, finally, the codification of the relevant knowledge that will have to be embedded in the KBE application. (Rocca, 2012).

2.1.1 Modeling tools (with KBE)

In order to apply the concepts of KBE a KBE system is required. A KBE system provides a framework for developing KBE applications which are flexible, detailed and dynamic domain knowledge models, supporting design of even very complex products. KBE-systems make the engineers able to write dedicated programs that perform complex and specific engineering task efficiently with the purpose of reducing product development time and cut engineering costs. E.g. before it took engineers weeks to iterate to a final design including geometry, simulations and documentation, with a KBE application this is now archived within days due to reuse of knowledge and design.

2.1.2 Modeling techniques

The top-down and bottom-up approach are both design strategies of information processing and knowledge ordering in software and product design and development. In the top-down approach the critical information is placed on a hierarchal top level and branches down to all lower component levels in the product.

Conversely, in the bottom-up approach, all base elements are modeled separately in detail and finally assembled into larger (sub-) assemblies.

A method to facilitate reuse is by dividing geometric transformation into two categories; morphological and topological. This way, the geometric models are given the knowledge which controls the geometrical transformations required for design automation. Morphological changes are manipulation of defined geometric entities inside an object such as shape, dimensions, position and orientation, e.g. it is enough to re-evaluate the instance. Topological changes are used to dynamically change and control the number and type of objects by adding, replacing or removing objects in the object tree, i.e. by using dynamic series of objects.

Morphological changes can be utilized in the model by making reusable design components from base elements. The component's design variable can be set on instantiation or inherited from parent components. The topological changes can be achieved by identifying the type and number of instances needed and dynamically attach them to the model. This approach is a modified top-down approach, referred to as dynamic top-down modeling, where the objects are linked to each other in a hierarchic relational structure.

The development of more general models with focus on flexibility and robustness, will allow engineers to work on a higher abstraction level where the use of low level CAD functions (i.e. points, lines, sweeps and extrusions) during the modeling and simulation phase is minimized if not fully eliminated. (Amadori, Tarkian, Ölvander, & Krus, 2012)

2.1.3 Design process automation

A classic approach in the engineering design process involves using design tools for component modelling and then specifics simulation tools for meshing, pre-processing, calculation, post-processing for the first design concept tested, and then it is necessary to start again for the next design modification. It is a methodical series of steps that often need to be repeated many times before production phase can be entered.

One aim of KBE tools and methods is to facilitate the connection between the design process and numerical simulations by integrating design and simulation tools in a collaborative environment. The purpose is to create a real link between the geometry of a component and the simulation context to automate the simulation task for the engineers to reduce time consumption spent on routine work.

KBE models with embedded design rules and engineering knowledge allows the geometry of a generic component to be modified from a set of parameters (Rocca, 2012). The transition from the geometric model to a numerical model is automated using an idealized model and mesh discretization to generate a finite element model. A finite element analysis (FEA) is done using integrated simulation software and the results are evaluated against specified criterions. The design optimization loop is run through several iterations by deriving a new set of parameters and generating new geometry. This streamlines the design process and allows engineers to test several component architectures very quickly and identify the main design concepts. (Roth, Chamoret, Badin, & Gomes, 2011)

2.2 Mechanisms

A mechanism is a collection of bodies (links) that may be interconnected by joints to constrain their relative motions. They form a mechanical system with one body fixed, with the purpose of altering a given input motion to produce a different output motion. The basic components of a mechanism can be characterized as follows:

- **Body (link):** A mechanical body is a physical component that usually is considered rigid. Its primary function is to hold fixed geometric relationships between its joint elements. Links can be subdivided into categories depending on the number of joints incident with that link.
- **Constraint (joints):** A joint is defined as a connection between two links. The primary kinematic function of a joint is to constrain the relative motions allowed between the connected bodies.

Mechanisms are in general divided into three categories, see Figure 2-1 :

- **Planar:** all moving points describe planar curves which all lie in parallel planes.
- **Spherical:** each moving body (or its extension) has one point that remains stationary as the system moves, and in which the stationary points of all bodies lie at a common location.
- **Spatial:** does not include any restrictions on the relative motions of their bodies.

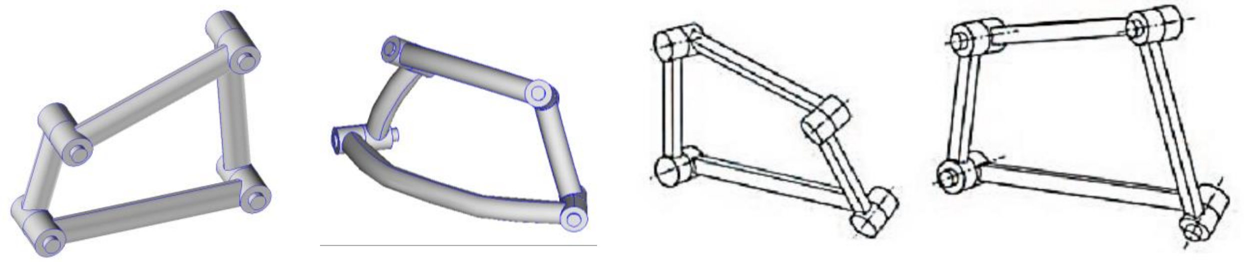
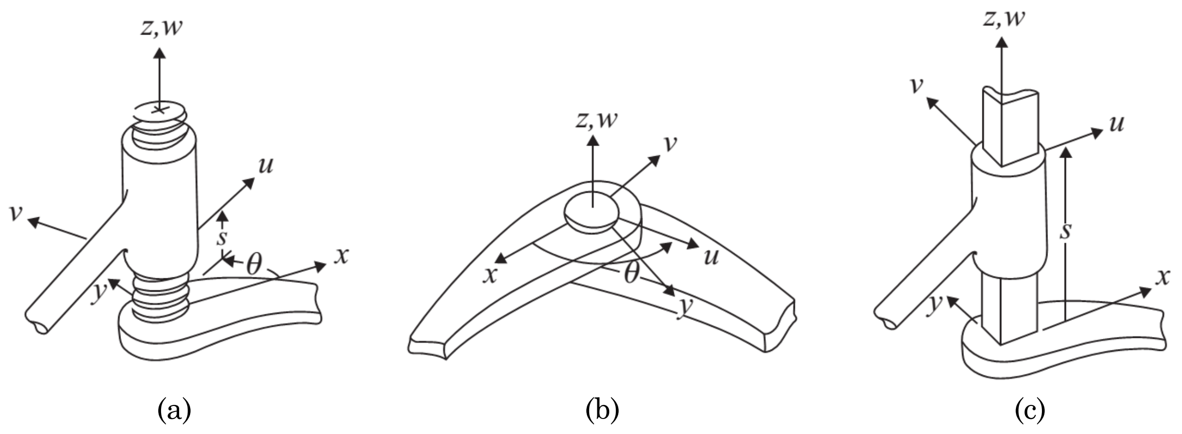


Figure 2-1 Four bar mechanisms: Planar, spherical, spatial (Bennett), and no relative motion

2.2.1 Constraints

Because a joint connects two mechanical bodies, the joint is not a separate physical entity in itself, but the interface composed of the contact surfaces on the two connected bodies. The two contact surfaces, when considered separately, are each referred to as a joint element and, when joined together, the surface of contact physically adds some constraint(s) to the relative motion between two links. The elements' shapes make it natural to refer to one of them as the solid/male element and the other as the hollow/female element. A coordinate system is placed in each element, and the relative motions permitted between the joint elements are assigned to variable parameters. Because these parameters are required as degrees of freedom in relative motion allowed by the joint, they are referred to as joint variables. As the mechanism moves, the linear transformation is described with a transformation matrix using functions of the joint variable(s).

Joint pairs are categorized into lower pairs and higher pairs. The six lower-pairs (Figure 2-2) have surface contact between their joint elements, whereas higher-pairs have line or point contact between their elemental surfaces. The lower pairs are described briefly as follows with the joint axis of a pair defined as the rotation, translation, or spindle axis, shown as the $\boldsymbol{z}$-axes of the joints in Figure 2-2.



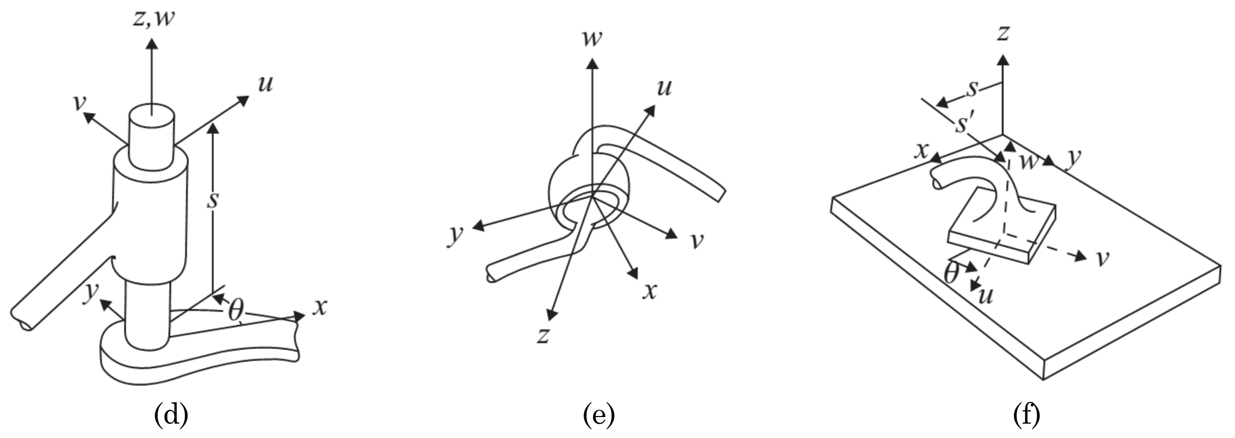


Figure 2-2 The six lower-pairs: (a) helical joint, (b) revolute joint, (c) prismatic joint, (d) cylindric joint, (e) spherical joint, (f) planar joint.

A *revolute* joint only allows rotation, between the paired elements, around a joint axis defined by the geometry. The revolute joint has only one degree of freedom (DOF) and the joint variable is the relative rotation ($\Delta\theta$) between the joint elements.

A *prismatic* joint only allows relative axial translation, of the paired elements with respect to each other, defined by the geometry of the joint. This type of joint also has one DOF, and the joint variable is the relative axial translation (Δs) between the joint elements.

A *cylindrical* joint is equivalent to a revolute joint in series with a prismatic joint with coaxial joint axes. It permits rotation about, and independent translation along, the joint axis defined by the geometry of the joint. Therefore, the cylindrical joint is a two DOFs joint and $\Delta\theta$ and Δs are both joint variables.

A *helical* joint allows two paired elements to rotate about, and translate along, the joint axis. However, the rotational angle $\Delta\theta$ and the axial distance Δs are related to the each other. Hence, the helical joint is a one DOF joint and either $\Delta\theta$ or Δs may be used as the joint variable defining the relative displacement of the elements.

A *spherical* joint is equivalent to three revolute joints intersecting at a central point and allows one element to rotate freely with respect to the other. Hence, the spherical joint has three DOFs and the joint variables may be chosen as three angles $\Delta\theta$, $\Delta\theta'$, and $\Delta\theta''$.

A *planar* joint is constituted of two planar surfaces, constrained to remain in contact but with two translational DOFs on a plane and a rotational DOF about the joint axis normal to the plane of contact. Hence, the plane joint is three DOFs joint and three joint variables can be chosen as Δs, $\Delta s'$, and $\Delta\theta$.

In addition to the six lower pairs, two more joint types are included as they give advantages in the simulation of mechanical systems. The *rigid* joint constrains all motion between two links, and is therefore used as a stiff connection with no joint variables and zero DOFs. The *open/free* joint has six unconstrained DOFs that do not provide any constraint on displacement or rotation the "connected" bodies. (Uicker et al., 2013)

2.2.2 Links

In general, the bodies of different types of mechanical systems come in an unlimited variety of shapes, sizes, mass, properties, and so on. The individual bodies making up a mechanism are called members or links. A member is a connection between two joints. Two or more members connected together such that no relative motion can occur between them will be considered as one link. The number of joints that are incident on a link determines its topological variation known as the degree of a link. Thus, a unary body has degree one, a binary body has degree two; a ternary body has degree three, and so on as shown in Figure 2-3.

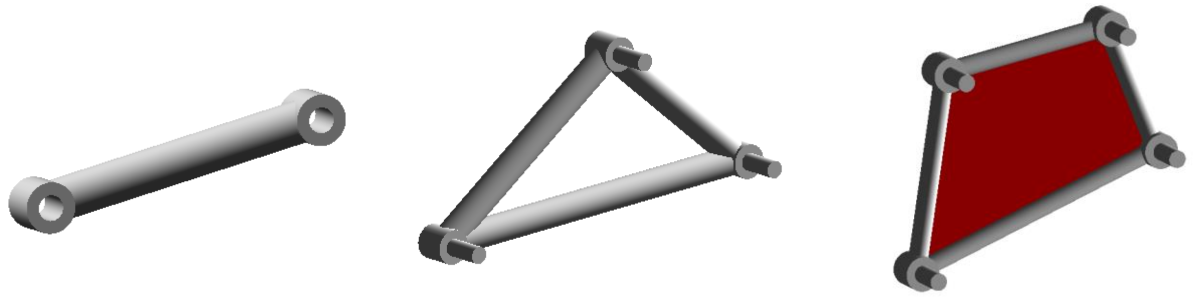


Figure 2-3 Example of binary, ternary and quaternary link

In kinematics, the study of relative motion among the various links of a mechanism, a body is considered rigid if its deformation under stress is negligibly small. Because masses are also neglected, the inertia effects and the forces that cause the motion are not taken into account.

However, for light-weight and high-speed mechanisms, the elastic effects of a mechanical body may become significant and must be taken into consideration. Deformations and flexibilities of mechanical bodies require a separate and comprehensive treatment, where each link has elastic (and also thermal) properties characteristic of its shape and material.

Dimensional synthesis deals with assigning detailed shapes, dimensions and material properties to the links in a mechanism. The shape is the path a member has between two constraints; usually it is a straight line for planar mechanisms and an arc for spherical mechanism, but it can have any kind of spatial configuration e.g. as seen in a car suspension system. When loading a mechanical system, the second moment of area of the cross section of a link is an important property that affects both the deflection and stress caused by compression, tension, shear, bending and torsion forces. The shape and cross section of the members on a link will determine the morphological variation of a link. To determine the exact dimensional properties it is desirable to perform a nonlinear analysis, which also takes elastic effects into consideration. But a good initial approximation can be obtained by calculating the forces in a truss system and applying classical beam theory.

2.2.3 Topology

The topological configuration of a mechanism can be expressed using an incidence table, oriented graph or an incidence matrix. They are useful formats representing the incidence relationships between links and joints, and the direction of a joint specifying which element is male and female. The fixed link is labelled zero.

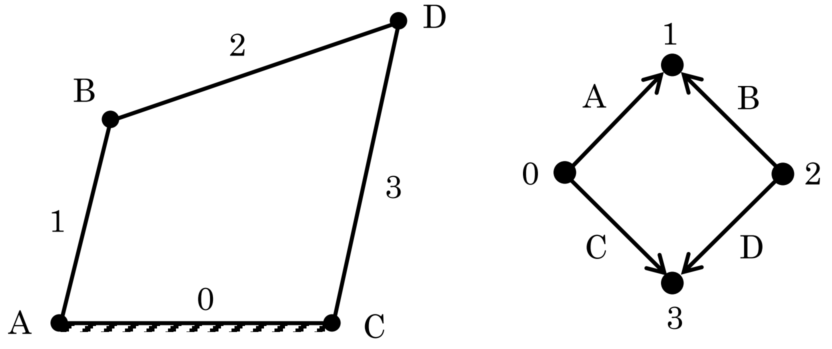


Figure 2-4 Structural (left) and graph representation of a four bar mechanism

Constraint	From link	To link
A	0	1
B	2	1
C	0	3
D	2	3

Table 1: Incidence table representing the mechanism in Figure 2-4

From the incidence relationships a topological analysis can be performed – studying the number of bodies, the number and type of joints, the pattern in which the bodies and joints are arranged, the number and pattern of closed loops[2], and other such characteristics that are determined by the connectivity of the system. These formats are applicable to a wide variety of mechanisms and suited for manipulation by a computer.

2.3 Parameterization of mechanisms

Sheth and Uicker (1971) present a generalized convention for measuring and describing the critical dimensional parameters of the components of a mechanical system. This is done by formulating mathematical models of the links and the constraints imposed on these links which in turn can be combined to form the overall system model on which various analysis procedures can be applied. Such a convention is called a kinematic convention and delivers a set of parameters that first and foremost can be used to parameterize the displacements which appear in the computational routines of kinematics. Bongardt (2013) shows that the Sheth–Uicker (SU) convention features advantages compared against the more popular Denavit–Hartenberg convention, covering the complexity of any mechanism, supporting systems containing multiple loops and with arbitrary constraints imposed on the links. The theory behind the SU convention is quite comprehensive and presented in detail here, because it is important for the context of work presented in later this thesis.

2.3.1 Representations of finite displacements

A frame F is used as a term for a local coordinate system, describing its rotation and translation relative to the origin. In kinematics, the axes of frames are ordered and interpreted according to a common principle:

1. The $\boldsymbol{z}$-axis is the major axis of F. It indicates the *dominant* direction of a frame. In case that F is attached to J, the $\boldsymbol{z}$-axis coincides with the joint axis.[3]
2. The $\boldsymbol{x}$-axis is the minor axis of F. It indicates the *secondary* direction of a frame. In case that F is attached to J, the configuration of J is indicated by the $\boldsymbol{x}$-axis.
3. The $\boldsymbol{y}$-axis is the redundant axis of F. Its direction follows from the right-hand rule, i.e., $\boldsymbol{y} = \boldsymbol{z} \times \boldsymbol{x}$.

In matrix notation a frame is given by

[2] if every link is connected to every other link by at least two distinct paths, the kinematic chain forms one or more closed loops

[3] E.g. the rotation, translation, or spindle axis.

$$\boldsymbol{F} = \begin{bmatrix} \boldsymbol{x} & \boldsymbol{y} & \boldsymbol{z} & \boldsymbol{p} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$ and $\boldsymbol{p}$ are elements of $\mathbb{R}^3$ and $\boldsymbol{p}$ is the location of the frame F.

The homogeneous matrix of a displacement $\boldsymbol{M}$ incorporates a linear rotation (via rotation matrix $\boldsymbol{R}$) and an affine linear translation (via the translation vector $\boldsymbol{t} \in \mathbb{R}^3$ that is linearized by the addition of the fourth dimension), denoted as

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ 0 & 1 \end{bmatrix}$$

Then, the spatial displacement matrix $\boldsymbol{M}$ between the two frames from $\boldsymbol{F}_D$ to $\boldsymbol{F}_A$ is expressed via

$$\boldsymbol{M} = (\boldsymbol{F}_D)^{-1} \cdot \boldsymbol{F}_A$$

The parametric form of the line, G, passing through two points $\boldsymbol{a}$ and $\boldsymbol{b}$, with $\boldsymbol{\omega} = \boldsymbol{b} - \boldsymbol{a}$, reads:

$$G = G(\boldsymbol{a}, \boldsymbol{\omega}) = \boldsymbol{a} + \boldsymbol{\lambda} \cdot \boldsymbol{\omega}$$

Where $\boldsymbol{p}$ denotes an anchor point of the line, and $\boldsymbol{\omega}$ denotes the direction vector of the line. The homogeneous form of a line, G, passing through two points a = (a_0, $\boldsymbol{a}$) and b = (b_0, $\boldsymbol{b}$) reads:

$$G = G(\boldsymbol{a}, \boldsymbol{b}) = (a_0\boldsymbol{b} - b_0\boldsymbol{a}, \boldsymbol{a} \times \boldsymbol{b}) = (\boldsymbol{\omega}, \boldsymbol{v_0})$$

Where vector $\boldsymbol{\omega}$ is referred to as the *direction* and vector $\boldsymbol{v}_0$ as the *orthogonal moment* of the line. The coordinates of the homogeneous form are also called Plücker coordinates.

Considering two lines, $G_{ij} = (\boldsymbol{\omega}_{ij}, \boldsymbol{v_0}_{ij})$ and $G_{jk} = (\boldsymbol{\omega}_{jk}, \boldsymbol{v_0}_{jk})$ their relative poses can conveniently be analysis using homogenous coordinates. In Table 2, the four possible poses of a pair of lines are compared.

Configuration	**Common Points**	**Directions**	**Distance**	**Special Points**
coincident	infinite	linear dependent	d = 0	anchor midpoint
parallel	none	linear dependent	d > 0	anchor midpoint projections
intersecting	one	linear independent	d = 0	intersection point

skew	none	linear independent	d > 0	closest points

Table 2 : Four relative poses (coincident, parallel, intersecting, skew) of two lines together with some characteristic entities. (Bongardt, 2013)

The following sequence of statements is used to determine the configuration of a pair of lines. They are coplanar (coincident, parallel, or intersecting) if and only if:

$$\frac{1}{2}\cdot\left(\boldsymbol{\omega}_{ij} * \boldsymbol{v}_{\mathbf{0}_{jk}} + \boldsymbol{v}_{\mathbf{0}_{ij}} * \boldsymbol{\omega}_{jk}\right) = 0$$

and coincident or parallell if:

$$\left|\boldsymbol{\omega}_{ij} \times \boldsymbol{\omega}_{jk}\right| = 0$$

and coincident if:

$$\left|(\boldsymbol{\omega}_{ij} \times \boldsymbol{v}_{\mathbf{0}_{ij}} - \boldsymbol{\omega}_{jk} \times \boldsymbol{v}_{\mathbf{0}_{jk}}) \times \boldsymbol{\omega}_{ij}\right| = 0$$

In case of skew or intersecting lines, for each line, one can find a point that has closest distance to the other line. For example, the closest point on G_{ij} to G_{jk} will be denoted by $\pi_{G_{ij}}(G_{jk})$ via the orthogonal projections for lines as

$$\pi_{G_{ij}}\left(G_{jk}\right) = \underset{\boldsymbol{p}_{ij}\in G_{ij}}{\operatorname{argmin}}\, dist(\boldsymbol{p}_{ij}, G_{jk})$$

For the case of intersecting lines, the closest points $\pi_{G_{ij}}(G_{jk})$ and $\pi_{G_{jk}}(G_{ij})$ coincide with the intersection point $G_{ij} \cap G_{jk}$. The distance of intersecting lines is *dist(*$\boldsymbol{p}_{ij}$*,* G_{jk}*) = 0*. If two lines, G_{ij} and G_{jk}, are given in parametric form as

$$G_{ij} = \boldsymbol{p}_{ij} + \boldsymbol{\lambda}_{ij} \cdot \boldsymbol{\omega}_{ij}$$

$$G_{jk} = \boldsymbol{p}_{jk} + \boldsymbol{\lambda}_{jk} \cdot \boldsymbol{\omega}_{jk}$$

Then the closest points $\pi_{G_{ij}}(G_{jk})$ and $\pi_{G_{jk}}(G_{ij})$, with $\boldsymbol{d}_{ij,jk} = \boldsymbol{p}_{jk} - \boldsymbol{p}_{ij}$ are given by

$$\pi_{G_{ij}}(G_{jk}) = \boldsymbol{p}_{ij} + \boldsymbol{\lambda}^{*}_{ij} \cdot \boldsymbol{\omega}_{ij} \qquad \boldsymbol{\lambda}^{*}_{ij} = \frac{(\boldsymbol{\omega}_{ij} \times \boldsymbol{\omega}_{jk}) \cdot (\boldsymbol{d}_{ij,jk} \times \boldsymbol{\omega}_{jk})}{(\boldsymbol{\omega}_{ij} \times \boldsymbol{\omega}_{jk})^2}$$

$$\pi_{G_{jk}}(G_{ij}) = \boldsymbol{p}_{jk} + \boldsymbol{\lambda}^{*}_{jk} \cdot \boldsymbol{\omega}_{jk} \qquad \boldsymbol{\lambda}^{*}_{jk} = \frac{(\boldsymbol{\omega}_{ij} \times \boldsymbol{\omega}_{jk}) \cdot (\boldsymbol{d}_{ij,jk} \times \boldsymbol{\omega}_{ij})}{(\boldsymbol{\omega}_{ij} \times \boldsymbol{\omega}_{jk})^2}$$

In case of coincident and parallel lines, neither $\pi_{G_{ij}}(G_{jk})$ nor $\pi_{G_{jk}}(G_{ij})$ can be determined from the geometry, since all points provide the same (minimal) distance to the other line. However, given the anchor points $\boldsymbol{p}_{ij}$ and $\boldsymbol{p}_{jk}$ of the lines, the midpoint $\boldsymbol{m}_{ij,jk}$ of those two $\boldsymbol{m}_{ij,jk} = \frac{1}{2}(\boldsymbol{p}_{ij} + \boldsymbol{p}_{jk})$ is determined. Then, the closest point of G_{ij} to this midpoint $\boldsymbol{m}_{ij,jk}$ is determined as $\pi_{G_{ij}}(\boldsymbol{m}_{ij,jk})$ and defined to be "the closest point" $\pi_{G_{ij}}(G_{jk})$ on G_{ij} to G_{jk}. For the other line G_{jk}, the computation works analogously.

A generalization of the closest point $\pi^*_{G_{ij}}(G_{jk})$ on G_{ij} to G_{jk} is defined as follows:

$$\pi^*_{G_{ij}}(G_{jk}) = \begin{cases} \pi_{G_{ij}}(G_{jk}) & \textit{if lines } G_{ij},\ G_{jk} \textit{ are intersecting or skew} \\ \frac{1}{2}\pi_{G_{ij}}(\boldsymbol{p}_{ij} + \boldsymbol{p}_{jk}) & \textit{if lines } G_{ij},\ G_{jk} \textit{ are coincident or parallel} \end{cases}$$

The generalized closest point $\pi^*_{G_{ij}}(G_{jk})$ on G_{ij} equals (1) the *closest point* on line G_{ij} in case of *skew* lines, (2) the *intersection point* in case of *intersecting* lines, (3) the *midpoint* of the *anchor points* in case of *coincident* lines, (4) the *projection midpoint* of the *anchor points* in case of *parallel* lines, see Table 2.

The generalized perpendicular direction given two lines G_{ij} and G_{jk} with directions $\boldsymbol{\omega}_{ij}$, $\boldsymbol{\omega}_{jk}$ and anchor points $\boldsymbol{p}_{ij}$, $\boldsymbol{p}_{jk}$, the $\boldsymbol{\omega}^*_{ij,jk}$ is then defined as follows:

$$\boldsymbol{\omega}^*_{ij,jk} = \perp^* (\boldsymbol{\omega}_{ij}, \boldsymbol{\omega}_{jk}) = \begin{cases} \boldsymbol{\omega}_{ij} \times \boldsymbol{\omega}_{jk} & \textit{if lines } G_{ij},\ G_{jk} \textit{ are intersecting or skew} \\ \tau_{\boldsymbol{\omega}_{ij}}(\boldsymbol{p}_{jk} - \boldsymbol{p}_{ij}) & \textit{if lines } G_{ij},\ G_{jk} \textit{ are coincident or parallel} \end{cases}$$

By means of the last definitions, it is possible to determine a 'shortest connection' between two lines with given anchors, independently of how these are opposed to each other.

2.3.2 Kinematic modeling convention

The relative axis orientation of kinematic joints located at two ends of the same link is invariable while in motion. Let Z_{ij} and Z_{jk} be two lines of joint axes that share one index j because the two joints are connect with the same link L_j. In this section, pairs of lines (of joint axes) and frames located on these are considered. Let an arbitrary chain of links which are connected by joints be fixed by L_i-L_j-L_k passing link L_j. Let $F_{(ij)_i}$, $F_{(ij)_j}$, $F_{ij\hat{k}}$, $F_{\hat{\imath}jk}$, $F_{(jk)_j}$ and $F_{(jk)_k}$ be the consecutive frames associated the chain of links. Then, for the frames that are attached to link L_j, the following short notation is defined:

$$F_{D_j} = F_{(ij)_j} \qquad F_{C_j} = F_{ij\hat{k}} \qquad F_{B_j} = F_{\hat{\imath}jk} \qquad F_{A_j} = F_{(jk)_j}$$

The 'hat' used in the triple index $ij\hat{k}$ can be read as a 'not'. For example: "the frame $F_{ij\hat{k}}$ belongs to the line tuple (Z_{ij}, Z_{jk}). It lies on the closest point of the line pair that does not lie on the line which contains the index k in its name".

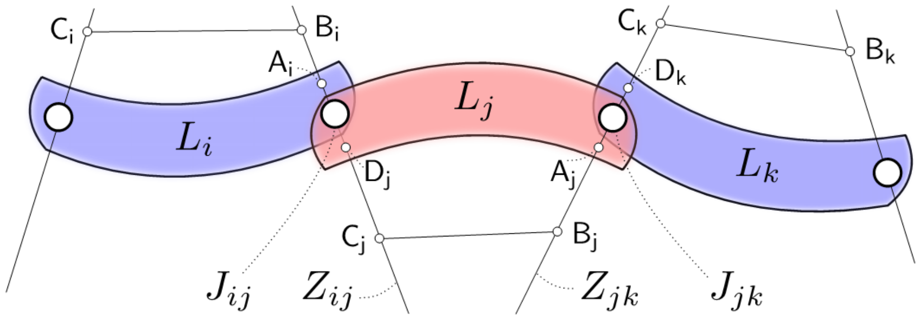


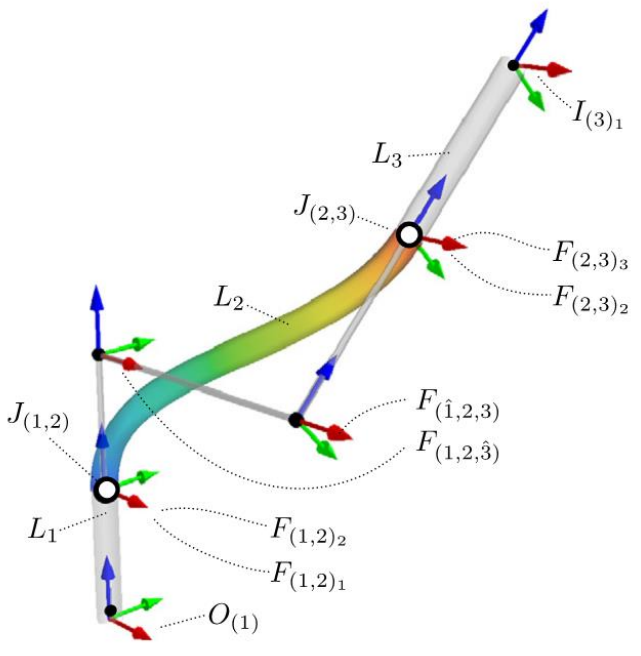
Figure 2-5 Line configurations of Z_{ij} and Z_{jk} for a chain of links L_i, L_j, L_k together with named frames.(Bongardt, 2013)

To define the constant shape-parameters for a link, the common perpendicular is found between the joint axes Z_{ij} and Z_{jk}. Initially each link has one coordinate system attached to each joint connected to the link. The frame F_{D_j}on link L_j in joint J_{ij} is an arbitrarily chosen coordinate system which includes the joint axis Z_{ij}, but without reference to common perpendicular. A second frame F_{A_j} is also defined, fixed in the constraint of joint J_{jk} on link L_j. It is chosen such its $\boldsymbol{z}$-axis lies along the joint axis of Z_{jk}. The $\boldsymbol{x}$-axes of the frames are chosen conveniently, e.g. according with the local geometry of the link they are attached to.

An augmented frame is placed in each of the closest points on Z_{ij} and Z_{jk}, with the $\boldsymbol{z}$-axis aligned along the joint axis and the $\boldsymbol{x}$-axis along the common perpendicular. The $\boldsymbol{y}$-axis is aligned to complete the right hand coordinate system. The geometric meaning of six parameters of the SU convention is given in Table 3 and visually presented in Figure 2-6.

Symbol	**Geometric Description**	**Alignment**
γ	const. *angular* distance of $\boldsymbol{x}_{D_j}$ and $\boldsymbol{x}_{C_j}$	around $\boldsymbol{z}_{D_j}$
c	const. *linear* distance of $\boldsymbol{x}_{D_j}$ and $\boldsymbol{x}_{C_j}$	along $\boldsymbol{z}_{D_j}$
β	const. *angular* distance of $\boldsymbol{z}_{C_j}$ and $\boldsymbol{z}_{B_j}$	around $\boldsymbol{x}_{C_j}$
b	const. *linear* distance of $\boldsymbol{z}_{C_j}$ and $\boldsymbol{z}_{B_j}$	along $\boldsymbol{x}_{C_j}$
α	const. *angular* distance of $\boldsymbol{x}_{B_j}$ and $\boldsymbol{x}_{A_j}$	around $\boldsymbol{z}_{B_j}$
a	const. *linear* distance of $\boldsymbol{x}_{B_j}$ and $\boldsymbol{x}_{A_j}$	around $\boldsymbol{z}_{B_j}$

Table 3: Geometric meaning of the six parameters of the Sheth – Uicker convention. The six parameters (γ, c, β, b, α, a) describe the link displacement.(Bongardt, 2013)



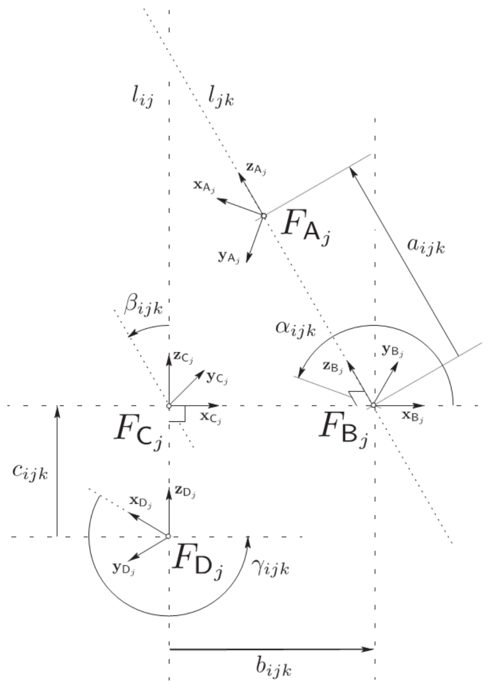


Figure 2-6 Example mechanism with named links, joints and frames. and parameters for a displacement between frames F_{Dj} and F_{Aj}. (Bongardt, 2013)

Below is the pseudo code of the method for calculating augmented frame placements for the SU convention.

(In) Two frames $F_{D_j} = F_{(ij)_j}$ and $F_{A_j} = F_{(jk)_j}$ at joints J_{ij} and J_{jk} and attached to link L_j.

(Out) Two augmenting frames $F_{C_j} = F_{ij\hat{k}}$ and $F_{B_j} = F_{\hat{\imath}jk}$ attached to L_j.

(⊥) The common perpendicular $\boldsymbol{x}^*_{ij,jk} = \perp^* \left(Z_{ij}, Z_{jk} \right)$ of lines Z_{ij} and Z_{jk} is computed.

(***p***) The location $\boldsymbol{p}_{C_j}$ and $\boldsymbol{p}_{B_j}$ of frames F_{C_j} and F_{B_j} are fixed to the closest points $\pi^*_{Z_{ij}}(Z_{jk})$ and $\pi^*_{Z_{ij}}(Z_{jk})$ of lines Z_{ij} and Z_{jk}.

(***z***) The ***z***-axes $\boldsymbol{z}_{C_j}$ and $\boldsymbol{z}_{B_j}$ of frames F_{C_j} and F_{B_j} are aligned along the ***z***-axis of $\boldsymbol{z}_{D_j}$ and $\boldsymbol{z}_{A_j}$ of frames F_{D_j} and F_{A_j}.

(***x***) Do the lines $\boldsymbol{z}_{C_j} = Z_{ij}$ and $\boldsymbol{z}_{B_j} = Z_{jk}$ of frames F_{D_j} and F_{A_j} share (at least) one common point?

Yes. (in case of coincident and intersecting $\boldsymbol{z}$-axes)

- The $\boldsymbol{x}$-axes $\boldsymbol{x}_{C_j}$ and $\boldsymbol{x}_{B_j}$ of frames F_{C_j} and F_{B_j} are aligned along the $\boldsymbol{x}$-axes $\boldsymbol{x}_{D_j}$ and $\boldsymbol{x}_{A_j}$ of F_{D_j} and F_{A_j}

No. (in case of parallel and skew $\boldsymbol{z}$-axes)

- The $\boldsymbol{x}$-axes $\boldsymbol{x}_{C_j}$ and $\boldsymbol{x}_{B_j}$ of frames F_{C_j} and F_{B_j} are aligned along the direction of the common perpendicular $\boldsymbol{x}^*_{ij,jk}$.

($\boldsymbol{y}$) The $\boldsymbol{y}$-axes $\boldsymbol{y}_{C_j}$ and $\boldsymbol{y}_{B_j}$ of frames F_{C_j} and F_{B_j} are aligned so that they complete the right-hand coordinate system.

2.4 Finite Element Analysis

Mechanical systems are complex arrangements of structural and mechanical components with different design purposes and mechanical behavior. The analysis of mechanisms presents many unique challenges depending on the type of applications, operating speed, external and internal loading of the components. The mechanical system may experience small or large deformations that could lead to a change of mechanism performance.

Nonlinear mechanism simulations are critical in understanding and solving flexible mechanical systems. Due to the complexity of the mechanical components it may be practical to handle such systems through a finite element (FE) approach. Effective time domain dynamic simulations of multibody systems in an FE environment have been described by Sivertsen (2001).

The Finite Element Method (FEM) is a numerical technique for finding approximate solutions to boundary value problems for partial differential equations (PDE). It partitions a geometric model into a number of smaller regions, called finite elements, and is able to solve PDEs on almost any arbitrarily shaped region.

To solve partial differential equations with the finite element method, three components are needed:

- a discrete representation of a region, i.e. a mesh
- a partial differential equation
- boundary conditions that link the equation with the region

2.4.1 Meshing

The collection of the finite elements that, as a sum, make up the entire region is called a mesh. Finding the numerical solution is then based on computing the solution on the smaller elements and then combining the partial solutions into a solution over the entire mesh.

The system is defined by components called elements which are joined together at discrete points called nodes defining the element geometry. Finite element size and shape are chosen according to the required accuracy of the results. Elements can have intrinsic dimensionality of one, two or three space dimensions as shown in Figure 2-7.

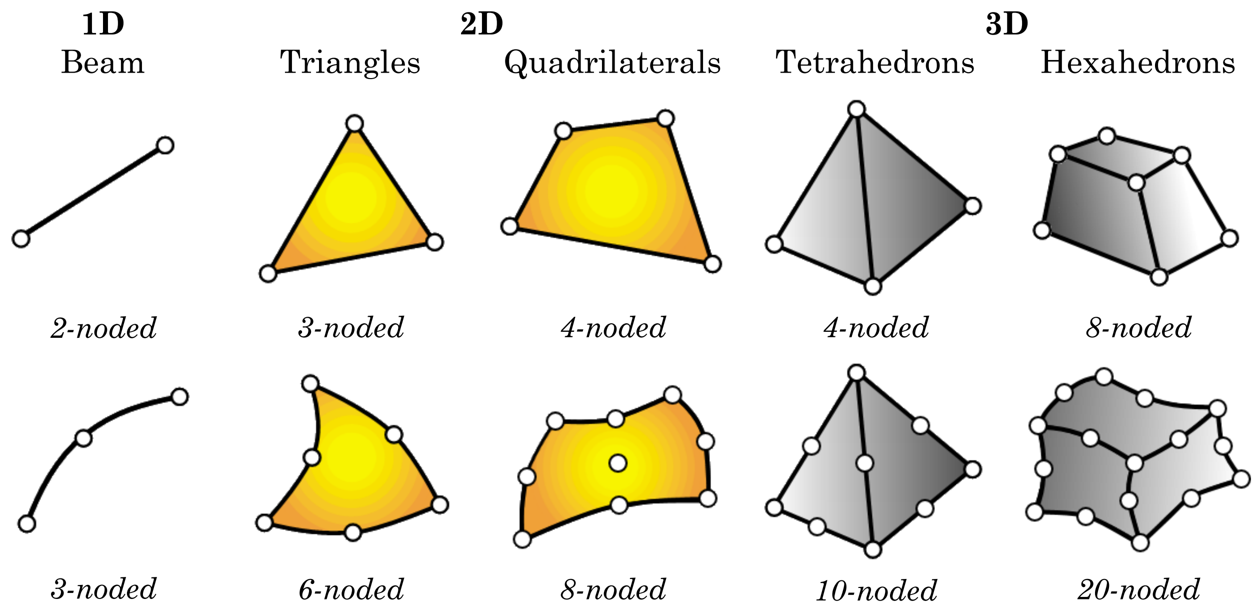


Figure 2-7 Typical finite element geometries in one through three dimensions.(Felippa, 2004)

A node is a coordinate location in space where the degrees of freedom (DOFs) are defined. The DOFs for this point represent the possible movement of this point due to the loading of the structure. The DOFs also represent which forces and moments are transferred from one element to the next. The results of a finite element analysis are usually given at the nodes.

2.4.2 Boundary conditions

In order to control the relative motions of a mechanical system boundary conditions between links are applied by constraining two or more points to be coincident throughout the simulation. Support constraints are used to restrain the structure against relative rigid body motion. This is done by attaching them to a "ground structure" which is viewed as the external environment. The resulting boundary conditions are often called *motion constraints*.

Multi Point Constraints (MPCs) are used to specify relations between DOF at different nodes to provide connectivity between substructures. These are functional equations that connect *two or more* displacement components. The degrees of freedom involved in each MPC are separated into master and slave freedoms. The slave freedoms are dependent to the master and then explicitly eliminated. The modified equations do not contain the slave freedoms.

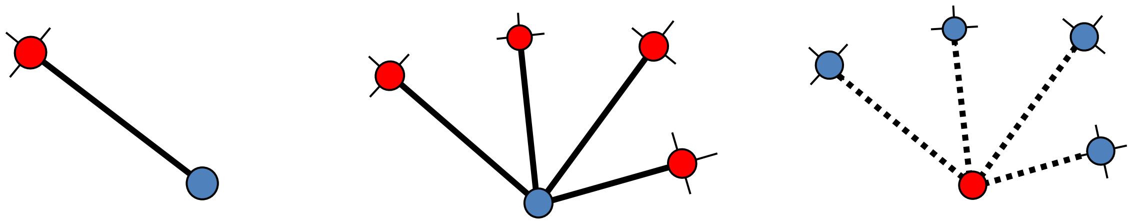


Figure 2-8 Common types of Rigid Body Elements, RBAR, RBE2 and RBE3.

Rigid Body Element, which is a form of MPC defining a rigid connection between the nodes in which nodes can be rigidly attached to each other. It adds an infinite stiffness to the model at these nodes. The three most common types of Rigid Body Elements are shown in Figure 2-8.

- **RBAR:** Rigid Bar with six DOF at each end
- **RBE2:** A MPC that defaults with six DOF for its independent node and the user can then choose what DOF to enforce upon the dependent nodes. The MPC creates a rigid mechanism between the independent and dependent nodes.
- **RBE3:** A MPC that defines the constraint behavior of the independent nodes as a function of its dependent nodes. The element has no default DOF and the user must choose the MPC's constraints carefully to prevent a free body mechanism. The MPC interpolates the overall constraint behavior of the dependent nodes upon the independent node. This formulation imparts no rigidly between the independent and dependent nodes. One key application is the smearing of a force from the independent node to the dependent nodes.

3 Methodology

This thesis is in the scope of the KBE area and the focus has been on the technical development of the KBE application. The emphasis on codification of knowledge about mechanism theory was done prior to this thesis and was mainly acquired from *A Digital Computer Based Simulation Procedure for Multiple Degree of Freedom Mechanical* (Sheth, 1972), V*irtual testing of mechanical systems, theories and techniques* (Sivertsen, 2001) and *Matrix Methods in the Design Analysis of Mechanisms and Multibody Systems* (Uicker et al., 2013). A summary of the research conducted was presented in the project thesis, *Introducing automation in design of mechanism* (Skaare, 2014).

3.1 Adaptive Modeling Language

TechnoSoft Inc. provides a KBE system using an object oriented programming language called AML (Adaptive Modelling Language). Originally written in Lisp (which is still the favored language in AI research and implementation), AML allows modeling the domain knowledge as a coherent network of classes. The AML paradigm provides a common interface to a solid modeler (Parasolid™) in addition to mesh generator (MSC Patran) and FEA solver (MSC Nastran).

AML was naturally chosen as the KBE-system for this thesis because Sivertsen closely collaborated with TSI during a sabbatical year in USA 2013-2014. He developed a pilot implementation in AML which was further studied and developed during the project thesis. The required software for this thesis was provided from TSI through Trapper Schuler.

3.2 Runtime environment

All development was committed on a 64-bit computer running Microsoft Windows 7 Home Premium (Service Pack 1) with AML Version 5.85.0 installed. Apart from the standard AML installation, the Analysis interface and Nastran interface modules, and some patches, were loaded separately (see Appendix C AML setup). The analysis module used was *aml-analysis-module-pack-type-3-01-06*, provided by TSI. This also includes the Nastran interface which manages all communication between an instance of an AML and the MSC Nastran software application. To work, Nastran must be installed and the path to nastran.exe-file and the directory for writing files relating to the analysis must be specified in the logical path file.

```
:nastran-path     "C:\Program Files\Siemens\NX 9.0\NXNASTRAN\bin"
```

```
    :nastran-data     "C:\Users\Rasmus\Documents\NTNU\Masteroppgave\nastran-
    data"
```

To standard way of running AML is by initiating the AML interpreter from an editor or directly from the command line. Interaction with the interpreter is done by writing commands and executing them with return. If an error occurs a pop-up dialog shows up, where the user can choose to “debug” or “abort”. The debugging feature is no longer supported and choosing the “debug” option only prints the stack trace to the console. When debugging an error, the error messages are often hard to decipher.

3.3 Editor

AML comes with a XEmacs editor included and at the center of XEmacs is the AML console. The console allows the developer to interact with the running AML process through a REPL (Read Eval Print Loop) interface in one of the XEmacs buffers. The developer is, amongst other, able to compile and run code, interact with the compiled models and send commands to the AML system. The XEmacs integration features Lisp-style syntax highlighting, as well as highlighting of some AML-specific expressions are available.

AML does currently not support any other editors, but Elstad and Granlund (2013) shows that integration with Sublime Text 2 is possible by making an REPL for AML, this was not compatible with AML 5.85.0.

3.4 Documentation

The AML Reference Manual (TechnoSoft, 2010) is a documentation including examples and explanation of different language constructs available in AML. It covers many but not all features in the AML package. TechnoSoft also delivers the AML Basic Training Manual (TechnoSoft, 2012). This manual is meant as a beginner’s course to AML by covering the most important features. The training manual thoroughly explains 11 exercises and came with a .zip file including source code for three supplement exercises and 7 additional examples. Together with the Reference Manual this is the only documentation provided by TSI. The Reference Manual is not available online, but bundled in the software. For convenience of the reader, a short summary is found in Appendix B. The full version is available through the Department of Engineering Design and Materials at NTNU.

When needed, Trapper Schuler from TechnoSoft has given support through email. Most problems have been solved through trial and error. When stuck, the inbuilt functions *describe*, *apropos* and *methods-for-class* are useful tools for finding class-names, function- and method arguments. The “inspect-tool” in the AML GUI works well when determining property-names.

3.5 Source code management

The source code was managed through the definition of a system named mechanism-system. Organizing the code into systems is a method for reusing code. Using systems also allows the code to be treated as a module that may be loaded, compiled, and archived as a single entity.

The definition of a system appears in the file named `system.def` using the define-system construct. The `system.def` must be in the directory returned when `(logical-path :mechanism-system)` is executed.

```
(define-system :mechanism-system
 :files '(
           "data-models.aml"
           "constraints.aml"
           "cross-sections.aml"
          "analysis.aml"
           "links.aml"
           "collections.aml"
          )
 )
```

For the system to be found for compiling or loading, the mechanism-system must be an entry in the logical path file.

```
:mechanism-system /code/mechanism-model/
```

The directory structure for the mechanism-system should then look like the following:

```
code/
     mechanism-model/
           system.def ;; File containing the system definition.
           sources/
                 data-models.aml ;; Source code file.
                 constraints.aml ;; Source code file.
                 cross-sections.aml ;; Source code file.
                 analysis.aml ;; Source code file.
                 links.aml ;; Source code file.
                 collections.aml ;; Source code file.
```

When the system is compiled the source code is read from the files specified in define-system. Compiling the system archives the source code with the binary files created, so editing an older version is possible by using the archived source code. The following compiles the system files

```
> (compile-system :mechanism-system)
```

And creates mechanism-system-1 subdirectory structure shown below.

```
MECHANISM-SYSTEM-1/
      system.def
      sources/
            data-models.aml ;; Archived source code file.
            constraints.aml ;; Archived source code file.
            cross-sections.aml ;; Archived source code file.
            analysis.aml ;; Archived source code file.
            links.aml ;; Archived source code file.
            collections.aml ;; Archived source code file.
      bins64/
            compilation.dat ;; Archived binary patch files
            data-models.btc64 ;; Archived binary file.
            constraints.btc64 ;; Archived binary file.
            cross-sections.btc64 ;; Archived binary file.
            analysis.btc64 ;; Archived binary file.
            links.btc64 ;; Archived binary file.
            collections.btc64 ;; Archived binary file.
```

When a system is loaded the binary files from the subdirectory that were created during the last compilation are loaded if no version number is supplied. If the source code is changed the changes will not be loaded until after a *compile-system* is performed. Running previously written AML-programs requires the system to be compiled or loaded first.

```
> (load-system :mechanism-system)
```

3.6 Programming style

AML follows the general common Lips coding conventions. Except from that lines with lone right parentheses are encouraged in the AML documentation.

The application has been created with focus on making reusable class features. The relationships between geometric elements and the hierarchical structure in the model was carefully planned, but also changed several times.

In true OOP style, objects are used for storing information. Lists consist of object references rather than static values and properties often contain references to other objects in the model tree.

Properties are declared as high up in the model tree as convenient. This makes use of *the* referencing in AML, which climbs the model tree looking for a value. Same with *default*, it will look up the tree for a property with the same name. If one is found, the value of that property is used. Otherwise, the specified default-formula is used. It is a common practice to organize common properties and objects of an AML model or class at one “place” within the model so other properties and objects can access them easily.

4 Development process

During the work with this thesis a KBE application has been developed demonstrating the main concepts, parametrization, modeling and meshing, proposed for automation in design of mechanism. The full source code of the application developed is found in Appendix D, and any references made to inbuilt class names or functions in AML are described in Appendix B.

4.1 Application input

An initial description of a mechanism is needed as input to the KBE application and to store the mechanism. The format had to be both user and computer readable, giving the user the option to introduce new mechanisms or choose and edit an existing design in the mechanism application.

4.1.1 Mechanism library

Each mechanism is stored in a separate folder in the mechanism library, located in `code/mechanism-model/library`. The application reads subfolders from the library and lets the user select a mechanism. Nodes, constraints and shapes are read from the selected mechanism.

4.1.2 Node list

All points of interest in the application are stored in `coordinates.txt`, a list of nodes containing xyz-coordinates. A node is considered a joint if it is connecting two links together. An unconnected node can be a design point for links, an application point for springs, dampers, application of forces or torques, or position, velocity and acceleration plots of coupler curves. Each node is stated as follows:

```
(x-coordinate y-coordinate z-coordinate)
```

Example input from the `coordinates.txt` file of the four bar mechanism:

```
0.0   0.0   0.0
0.0   0.15  0.0
0.3   0.0   0.0
0.3   0.375 0.0
0.6   0.6   0.0
```

4.1.3 Constraint list

The incidence relationships between links and joints are stored in `constraints.txt`, an expanded incidence table which also includes joint-type, joint-axis direction and additional joint variables. This format was chosen because it is list based an easily read from the application and edited by users compared to an incidence, graph or matrix, representation. Each constraint is stated as follows:

```
(node-index joint-type (link-incidence-list) (joint-direction) (joint-
variables))
```

Example input from the `constraints.txt` file of the four bar mechanism:

```
0     "revolute"  "(1 0)"     "(0 0 1)"   "()"
1     "revolute"  "(1 2)"     "(0 0 1)"   "()"
2     "revolute"  "(3 0)"     "(0 0 1)"   "()"
3     "revolute"  "(3 2)"     "(0 0 1)"   "()"
```

The *link-incidence-list* is specified as *(list from-link to-link)*. Each joint is a connection between two links and the order specifies which link is connected to the male-element *(from-link)* and the female-element *(to-link)* of the joint.

4.1.4 Shape-file

For each of the members composing a link, dimensional properties are stored in `shapes.txt`. With a reference to member on a link or specifying default properties, cross section type, dimensions and optional design points are defined. Each shape is stated as follows:

```
(link-index member-index (cross-section-type) (shape-dimensions)
(points-list) (weight-list))
```

Example input from the `shapes.txt` file of the four bar mechanism:

```
2          1          "nil"      "(0)"            "()"
default    default    "line"     "(0.02 0.02)"    "()"
```

4.1.5 Data models

Some basic data-models are defined to utilize the object oriented nature of AML. Point-, vector- and frame-objects are already defined in the AML package, but are extended to take advantage of object referencing. E.g. a vector is defined from one point-object and a direction and a frame is defined from one point-object and two vector-objects. When several objects refer to the same point-object, all objects are automatically updated if this value changes.

4.1.6 Collections

All information read from file is stored in collections as subobjects created using series-object. Coordinates are stored as point-objects, joints are stored as joint-models and shapes are stored as shape-model. A folder collection is also created with reference to all the subfolders in the library. From the unique link-references found in the incidence table, a collection of links is created. Each link has references to the joint elements which it is connected to. The topology of the mechanism is determined by the number and connectivity of the links and joints, and can be change by providing a different set of input to the application.

4.2 Initial frame placements

In the SU convention, the joint elements are related to each other by a coordinate system (frame) in each element. For lower pairs, the origins or $\boldsymbol{z}$-axis of the frames are coincident and describes the relative motion between them. The frames are arbitrary chosen but related by the joint variables.

While working with this thesis it became apparent that the physical part of link is not always connected to the joint center. An extra frame is therefore added to separate between the displacement caused by the solid geometry of the joint and the link. Bongardt (2013) inconspicuously mentions to include a joint displacement, but it is uncertain if they consent. This frame, constituting a construction frame, is used as a reference for the shape of a link and a represent the displacement caused by the interface of the joint element. When calculating the parameters of the SU convention, the construction frame is used as reference for the frames $\boldsymbol{F}_{Dj}$ and $\boldsymbol{F}_{Aj}$.

The lower kinematic pairs are all characterized by a joint axis which describes the axis of rotation and/or translation. A *general-joint-solid-model* is made containing the joint information read from input, the male and female element are both subobjects oriented according to the initial joint variables. Their $\boldsymbol{x}$-axis are directed towards the next joint on the first connected member.

```
(define-class master-joint-model
      :inherit-from (general-joint-solid-model)
      :properties (
            ;;
            )
      :subobjects (
            (male-element-solid-model :class (read-from-string (concatenate
            !constraint-type "-male-element"))
                  )
            (female-element-solid-model :class (read-from-string (concatenate
            !constraint-type "-female-element"))
                  )
            )
      )
```

Both joint elements, includes two frames, a main frame, located in the joint center, and sub-frame specifying where the construction of the link starts.

```
(define-class joint-element-model
      :inherit-from (general-joint-solid-model)
      :properties (
            gender (read-from-string (remove "-element-solid-model" (write-to-
            string (object-name !superior))))

            gender_int (case ^gender ('male 0) ('female 1) )
            link-incidence (nth ^gender_int ^^link-incidence)
            link-ref-incident-on-constraint (get-link-ref ^links ^link-
            incidence)
            )
      :subobjects (
            (main-frame :class 'frame-data-model
                  )
            (sub-frame :class 'sub-frame-model
                  )
            )
      )
```

The position of the sub-frame is calculated from the geometry of the joint, see 4.4 Joint geometry. Examples of frame placements for a revolute joint are shown in Figure 4-1. Note that the main-frame and sub-frame are coincident for the female element since the joint center and link start in the same position.

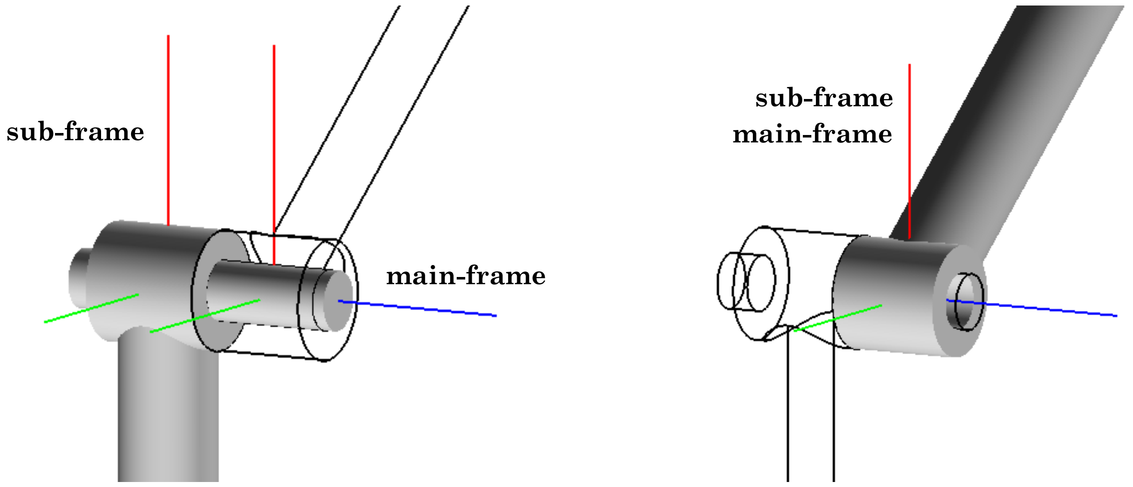


Figure 4-1 Frame placements on male element (left) and female element of revolute joint

4.3 Link geometry

All links are considered as a connection between joints. They can be characterized as a collection of members, where each member connects two joints. A link will have a topological configuration determined by the degree of the link. While the shape and dimensions of the members will determine the morphology

The joints incident on a link are collected from the joint collection. Each link then has a number of joints n, attached to it. For a binary link (n=2) there is only one possible connection between the two joints. For a ternary link (n=3) there are three possible connections between the three joints. Calculating all possible connections between n numbers of joints is a complete graph problem. The complete graph on n joints is denoted by K_n. The number of possible connections is given by $\frac{n}{2}(n-1)$.

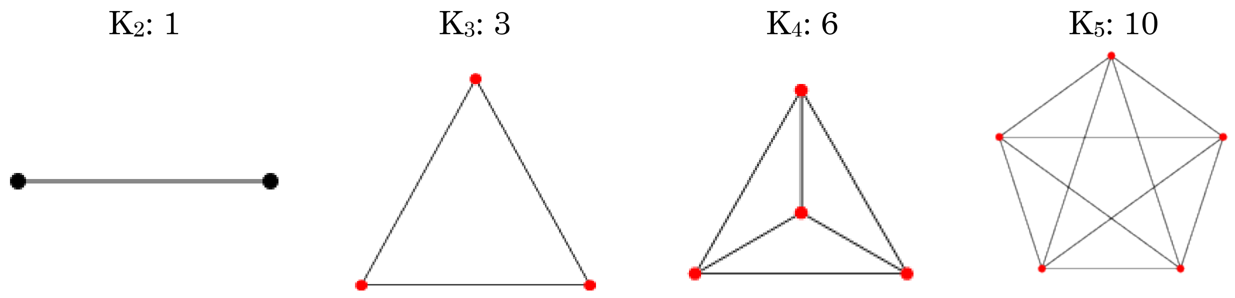


Figure 4-2 Number of possible connections between joints

Each edge in the graph K_n is equivalent to a member on the link connecting two joints. For the link to be connected each joint has to be connected to at least one other joint. Not all connections on a link are necessary, for K_3 and higher the user can select the preferred connections. The possible connections are calculated using the following algorithm, taking a list of joints on a link as argument.

```
(defun list-combinations (p)
      (let (
            (l (if (typep p 'list)
                     p
                  (if (typep p 'fixnum)
                        (loop for i from 0 to (1- p) collect i)
                     (list ) ) ) )
            (n (length l))
            )
            (loop for j from 0 to (- n 2)
                  append (loop for k from (1+ j) to (1- n)
                        collect (list (nth j l) (nth k l))
                        )
                  )
            )
        )
```

For each link the joints incident on that link are collected looping through the joint collection. And the possible joints combinations for the members are calculated using *list-combinations*.

```
constraints-incident-on-link-list (loop for kid in (the constraints-ref-list)
      for con = (get-constraint-incidence kid ^link-index)
      when con collect con
      )

connection-between-2-constraints-combinations (list-combinations ^constraints-
incident-on-link-list)
```

Below is an example of the return values from *list-combinations* with joint indexes as argument. With four joints there are six possible connections between the joints 0, 1, 2 and 3.

```
      > (list-combinations (list 0 1 2 3))
      ((0 1) (0 2) (0 3) (1 2) (1 3) (2 3))
```

4.3.1 SU parameters

For each member, its joint references are selected according to its index on a link.

```
      connection-between-constraints (nth ^index ^^connection-between-2-
      constraints-combinations)
```

The reference to the sub-frame in each joint element is set.

```
      frame_D (the sub-frame (:from (nth 0 ^connection-between-constraints) ))
      frame_A (the sub-frame (:from (nth 1 ^connection-between-constraints) ))
```

The parameters for line Z_{ij} and Z_{jk} are extracted from the frames.

```
      pij (convert-coords ^frame_D '(0 0 0) :from :local :to :global)
      wij (convert-vector ^frame_D '(0 0 1) :from :local :to :global)

      pjk (convert-coords ^frame_A '(0 0 0) :from :local :to :global)
      wjk (convert-vector ^frame_A '(0 0 1) :from :local :to :global)
```

The SU convention is implemented in AML from the theory presented by Bongardt (2013). The line configuration, the closest point on each line and the common perpendicular are calculated from the corresponding methods. The augmented frames are placed before calculating the SU parameters. An extract is shown below, and the full source code is found in Appendix D.

```
(define-class connection-model
      :inherit-from ()
      :properties (
            line-config (line-pose (the superior))
            inter_points (inter_section (the superior) ^line-config )
            perpendicular-dir (perp-dir (the superior) ^line-config )

            param_a (vector-length (subtract-vectors (the origin (:from
            ^frame_A)) (the origin (:from ^frame_B))))

            param_b (vector-length (subtract-vectors (the origin (:from
            ^frame_B)) (the origin (:from ^frame_C))))

            param_c (vector-length (subtract-vectors (the origin (:from
            ^frame_C)) (the origin (:from ^frame_D))))
            )
      :subobjects (
            (frame_B :class 'frame-data-model
                  (point-ref :class 'point-data-model
                        coordinates (nth 0 ^^^inter_points)
                        )
                  (z-vector-ref :class 'vector-data-model
                        direction ^^wij
                        )
                  (x-vector-ref :class 'vector-data-model
                        direction ^^perpendicular-dir
                        )
                  )

            (frame_C :class 'frame-data-model
                  (point-ref :class 'point-data-model
                        coordinates (nth 1 ^^^inter_points)
                        )
                  (z-vector-ref :class 'vector-data-model
                        direction ^^wjk
                        )
                  (x-vector-ref :class 'vector-data-model
                        direction ^^perpendicular-dir
                        )
                  )
            )
      )
```

4.3.2 Connection shape

The shape of a member connecting to constraints is represented using a spline curve. Splines are commonly used in computer graphics for generating and representing curves and surfaces, and work by interpolating control points to obtain smooth continuous functions. The start and end points of the curve is given by F_{Dj} and F_{Aj}. The points in-between are either automatically generated or given by design points in the Shape-file (see 4.1.4). This makes it is possible to achieve a high degree of flexibility, but still generates a simplified curve when no extra control points are specified.

NURBS (Non-Uniform Rational Basis Splines) generalizes both B-splines and Bézier curves and is given by *nurb-curve-object* in AML. Like B-splines, they are defined by their order, and a knot vector, and a set of control points determining the shape of the curve, but unlike simple B-splines, the control points each have a weight. When the weight is equal to 1, a NURBS is simply a B-spline.

An additional frame is placed at the start and the end of the connection, with the ***x***-axes tangential to the spline curve. These will be used for positioning the cross section of a member. The *nurb-curve-object* is given in the *connection-model* which is also used for calculating the SU parameters seen below:

```
(define-class connection-model
      :inherit-from (nurb-curve-object)
      :properties (
            start-point (the origin (:from ^frame_D))
            end-point (the origin (:from ^frame_A)))

            start-weight (list (append ^start-point (list 1)))
            end-weight (list (append ^end-point (list 1)))

            points (let (
                        (shape-points (loop for p-index in (the point-list
                        (:from ^shape-ref))
                              for w in ^weight-points
                              collect (append (the coordinates (:from (nth p-
                              index ^point-ref-list)) ) (list w)) ))
                              )
                  (if shape-points
                        (append ^start-weight shape-points ^end-weight)
                     (append ^start-weight ^middle-points ^end-weight)
                     )
                  )
            ;; automated points are given in middle-points
            )
      :subobjects (
            (spline-frame_start :class 'frame-data-model
                  )
            (spline-frame_end :class 'frame-data-model
                  )
            )
```

```
        )
```

4.3.3 Shape parameterization

The SU parameters allows for a flexible convention describing the linear displacement of over a link. An interesting problem is to create geometry of a link using these parameters. There is not any research found in using the SU parameters to generate geometry, but some key concepts are presented here.

For a planar mechanism, where the *z*-axes of the joints are parallel, design points are generated at half distance along the common perpendicular from each sub-frame. This creates a straight line for joints with coincident xy-planes and also work for any other parallel configuration with the parameters *a* and *c* greater than 0 shown in Figure 4-3.

```
(let (
      (start-tangent (add-vectors ^start-point
            (multiply-vector-by-scalar (normalize ^perpendicular-dir) (half
      ^param_b) ) ))
      (end-tangent (add-vectors ^end-point
            (multiply-vector-by-scalar (normalize ^perpendicular-dir) (- (half
      ^param_b)) ) ))
      (start-weight (list 0.5))
      (end-weight (list 0.5))
      )
      (if (roughly-same-point start-tangent end-tangent)
            (list (append start-tangent start-weight) )
         (list (append start-tangent start-weight) (append end-tangent end-
         weight) )
         )
      )
```

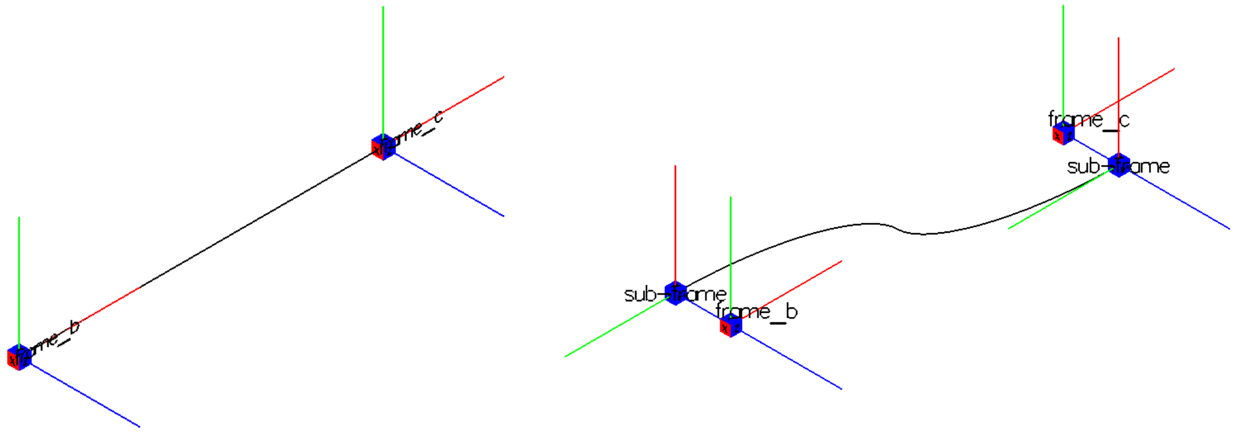


Figure 4-3 Generated spline curves of planar mechanism, joint axes in blue

In case of intersecting joint axes where both joints lie on the same circle, the NURBS curve can be used to approximate an arc[4] shown in Figure 4-4. Below is a possible way of integrating this in the application.

```
(let (
      (center (nth 0 ^inter_points ))

      (middle-point (add-vectors center (multiply-vector-by-scalar (normalize
      (add-vectors (subtract-vectors ^start-point center) (subtract-vectors
      ^end-point center ))) ^param_a) ) )

      (angle-start-middle (/ (angle-between-2-vectors (subtract-vectors
      ^start-point center) (subtract-vectors middle-point center )) 2 ))

      (start-tangent (add-vectors center (multiply-vector-by-scalar (normalize
      (add-vectors (subtract-vectors ^start-point center) (subtract-vectors
      middle-point center ))) (/ ^param_a (cosd angle-start-middle))) ) )

      (angle-middle-end (/ (angle-between-2-vectors (subtract-vectors middle-
      point center) (subtract-vectors ^end-point center )) 2))

      (end-tangent (add-vectors center (multiply-vector-by-scalar (normalize
      (add-vectors (subtract-vectors middle-point center) (subtract-vectors
      ^end-point center ))) (/ ^param_a (cosd angle-middle-end))) ) )

      (start-weight (list (sind (/ (angle-between-2-vectors
            (subtract-vectors start-tangent ^start-point)
            (subtract-vectors start-tangent middle-point)
            ) 2) ) ) )
      (middle-weight (list 1))
      (end-weight (list (sind (/ (angle-between-2-vectors
            (subtract-vectors end-tangent middle-point)
            (subtract-vectors end-tangent ^end-point)
            ) 2) ) ) )
      )
      (list (append start-tangent start-weight) (append middle-point middle-
      weight) (append end-tangent end-weight) )
      )
```

[4] http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/NURBS/RB-circles.html

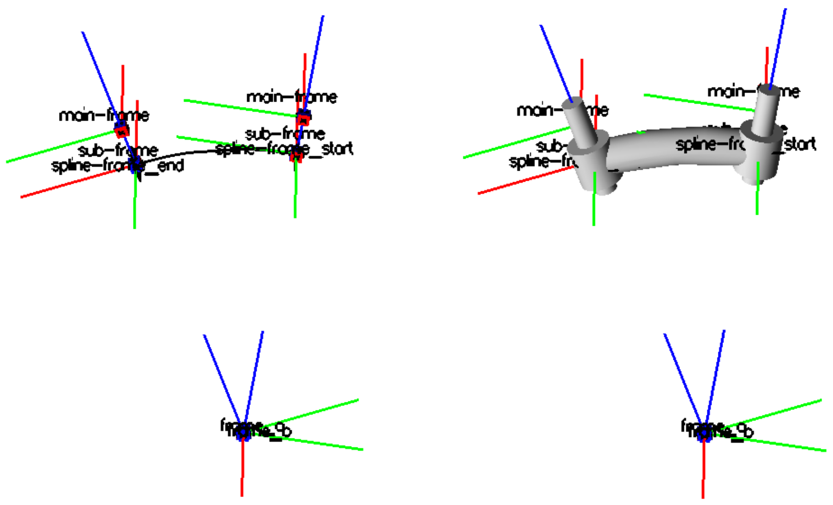


Figure 4-4 Generated spline curve and geometry for spherical mechanism, joint axes in blue

In case of coincident or skew configuration there is no straight forward solution. The coincident joints can be connected by a half circle or a straight line while the skew joints can be connected by any spatial shape. It is therefore more convenient for the user to specify design point to ensure the desired shape.

4.3.4 Sweep

To make a solid geometry of a member the splines are used as guides for sweeping cross sections between the constraints, creating a sweep-object using *general-sweep-class*. Any desired cross section can be used, typical types are included but additional can be added by inheriting from *section-model*. The code for the circular cross section is shown below. Note that if a cross section is symmetric, an additional point must be imprinted on the section-curve for the cross section at each end creating and extra vertex that matches when sweeping.

```
(define-class circular-section
  :inherit-from (imprint-class section-model)
  :properties (
               target-object ^disc
               tool-object-list (list ^p1)

               diameter (average ^width ^height)
               (disc :class 'disc-object
                   diameter ^^diameter
                   )
               (p1 :class 'point-object
                   coordinates (list (/ ^diameter 2) 0 0)
                   )
               )
```

```
        )
```

A cross section is placed at the start and end of each guide. The start and end cross section has a reference to the frames placed at the start and end points. This allows for creating a variational sweep, often seen on different links in mechanisms such as a car suspension.

Below is code extract from general sweep class:

```
(define-class member-solid-model
      :inherit-from (general-sweep-class)
      :properties(
            ;;; sweep parameters
            profile-objects-list (list
                  ^cross-section_D
                  ^cross-section_A
                  )
            path-points-coords-list (list
                   (the origin (:from ^frame_D))
                   (the origin (:from ^frame_A))
                  )
            profile-match-points-coords-list (list
                  (vertex-of-object ^cross-section_D)
                  (vertex-of-object ^cross-section_A)
                  )
            )
      :subobjects (
            (connection :class 'connection-model
                                  )
            (cross-section_D :class !cross-section-type
                  reference-object (the spline-frame_start (:from
            ^connection))
                  orientation (list
                        (rotate 90 :x-axis)
                        (rotate 90 :z-axis)
                        )
                  )
            (cross-section_A :class !cross-section-type
                  reference-object (the spline-frame_end (:from ^connection))
                  orientation (list
                        (rotate 90 :x-axis)
                        (rotate 90 :z-axis)
                        )
                  )
            )
      )
```

The type of class instantiated for the cross sections is treated as a variable, meaning the object type can change dynamically during runtime. The class *option-property-*

class is used for properties whose value is equal to one element of a list of available options. The cross section type is represented with an option menu, the list of names of classes that directly inherit from the section-model are used as options. If a new cross section is defined and inherits from section-model, it will be an option in the list.

Each sweep-object has a property called cross-section-type declared as follows.

```
(cross-section-type :class 'option-property-class
      label "Cross-section Type"
      mode 'menu
      formula (nth 0 !options-list)
      options-list (reverse
                  (class-direct-defined-subclasses 'section-model)
            )
      labels-list (loop for option in !options-list
                  collect (remove "-section" (write-to-string option))
            )
)
```

A few different examples of cross sections are shown in Figure 4-5 where the start and end cross section matches for different orientations and sizes. This works for any new cross-section added. Pro tip: When creating a cross section using *difference-object*, specify *simplify? t*, or sweeping will not work.

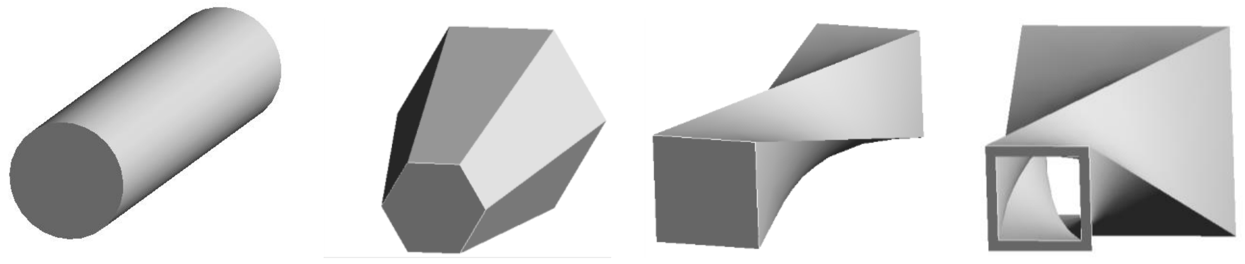


Figure 4-5 Sweeps with different varying orientation and different sized cross section

4.3.5 Surfaces

Possible combinations of 3 connections to create surface, check against visible members that are available.

If three splines are connected creating a loop it is possible to create a surface using *surface-from-three-edge-curves-class*. The number of possible surfaces are given by $\frac{n}{6}(n-2)(n-1)$, where n is the number of constraints on a link. To possible combinations of splines was calculated using these algorithms, taking the number of joints as argument.

```
(defun list-3-subset-combinations (p)
     (let (
          (l (if (typep p 'list)
               p
               (if (typep p 'fixnum)
               (loop for i from 0 to (1- p)
                    collect i) (list ) ) ) )
          (n (length l))
     )
     (loop for i from 0 to (- n 3)
          append (loop for j from (1+ i) to (- n 2)
               append (loop for k from (1+ j) to (1- n)
                    collect (list (nth i l) (nth j l) (nth k l))
                    )
               )
          )
     )
   )
```

The first function calculates the combinations of three constraints creating a loop.

```
> (list-3-subset-combinations 4)
((0 1 2) (0 1 3) (0 2 3) (1 2 3))
```

For a link with 4 joints (see K_4 in 4.3) there are 4 possible combinations of the joints 0, 1, 2 and 3. But it is the combination of splines connected that has to be calculated.

```
 (defun list-combinations (p)
     (let (
          (l (if (typep p 'list)
               p
               (if (typep p 'fixnum)
               (loop for i from 0 to (1- p)
                    collect i) (list ) ) ) )
           (n (length l))
          )
          (loop for j from 0 to (- n 2)
               append (loop for k from (1+ j) to (1- n)
                    collect (list (nth j l) (nth k l))
               )
          )
     )
)
```

The function *list-combinations* (given in 4.3) is used to calculated the possible combinations of the three constraints given from *3-edge-loop-combinations*. Then the below function checks for the occurrence of this combination in the *connection-combinations* and returns the position that equals the splines' index.

```
(defun sweep-loop-combinations (n)
      (let (
            (c-loops (3-edge-loop-combinations n) )
            (sweep-con (connection-combinations n) )
            )
            (loop for ci from 0 to (1- (length c-loops))
                  for list-com = (list-combinations (nth ci c-loops))
                  collect (loop for si from 0 to (1- (length list-com))
                        collect (position (nth si list-com) sweep-con)
                  )
            )
      )
)
```

The above function calculates the position of a constraint combination and returns the indexes of the connected three splines creating a surface. Below is an example of combinations of three splines that could create a surface.

```
> (sweep-loop-combinations 4)
((0 1 3) (0 2 4) (1 2 5) (3 4 5))
```

To make sure only members that are displayed will be used to create a surface their indexes are collected and the combinations are calculated and compared to the possible configurations.

```
visible-members-index (loop for mem in ^visible-members-ref-list
      collect (the index (:from mem))
      )

valid-surface-loops (intersection ^closed-loops-combinations (list-3-
subset-combinations ^visible-members-index))
```

The possible surfaces are then created using series-object and a thickness is added through *surface-thickened-class*.

4.4 Joint geometry

The joints are stored in the constraint collection and are not defined on the links. Instead each joint element has a reference to a link, and each link to its joint elements. To model the solid geometry of the joint, the dimension is determined from the members connected to the joint element and its pair element.

The reason for this is based on the idea that the joints are transferring the forces between links, if the links are sufficiently sized the joints can be expresed as a function of the link's dimension. For the demonstrational purpose of this thesis the joint elements are chosen to be a factor bigger than the link. An exact funtional relationship between link and joint dimensions can be derived and later implemented in the application.

The sub-frame is translated away along the joint axis according to the joint dimension to allow for space between links. The position of the sub-frame is dependent on joint and element type.

A *local-max-dimension* of a joint is found by looping through all members connected to a joint element, if they are currently displayed. It checks if the joint is connected to the start or end cross section, and collectes the maximum height and width of the members connected. *Local-max-dimension* are calculated as follows.

```
(loop for member in (the members-ref-list (:from (the link-solid-geometry
(:from ^link-ref-incident-on-constraint))))

      for pos = (position !superior (the connection-between-constraints (:from
      member)))

      when (and pos (the display? (:from member)) )
      maximize (if (= pos 0)
            (max-width (the cross-section_D (:from member)) )
            (max-width (the cross-section_A (:from member)) ) )
            into max-w
      and  maximize (if (= pos 0)
            (max-height (the cross-section_D (:from member)) )
            (max-height (the cross-section_A (:from member)) ) )
            into max-h

      finally (return (list max-w max-h))
      )
```

The maximum width and height of the local and the paired element is then used as parameter for determining the dimensions of the joint and will also be used to determine the placement of the sub-frames.

```
max-width (* ^scale-factor (max
      (first (the local-max-dimension (:from ^female-element-solid-model)))
      (first (the local-max-dimension (:from ^male-element-solid-model)) ) )
      )

max-height (* ^scale-factor (max
      (second (the local-max-dimension (:from ^female-element-solid-model)))
      (second (the local-max-dimension (:from ^male-element-solid-model)) ) )
      )
```

4.5 Assembly

Since the joint elements in reality are connected to a link, they should be assembled together with the other members and the surfaces. This is done by collecting the parts of a joint element which are union or difference into separate lists.

```
union-list (loop for l in ^constraints-incident-on-link-list
      append (the union-list (:from l))
      )
difference-list (loop for l in ^constraints-incident-on-link-list
      append (the difference-list (:from l))
      )
```

Each link then collects the objects from its connected elements and assembels them using a union-object. At last a difference-object is created. This is to make sure the hollow parts of joint elements are subtracted from the link where necessary.

```
(union-element :class 'union-object

      object-list (append
            ^surfaces-ref-list
            ^visible-members-ref-list
            ^^union-list
            )
      simplify? t
      )

(difference-element :class 'difference-object

      object-list (append
            (list ^union-element)
            ^difference-list
            )
      simplify? t
      )
```

To prepare the geometry for meshing, periodic or symmetric surfaces need to be imprinted, this is done with the *geometry-with-split-periodic-faces-class*.

Each link is separated in to three models, the geometric model, the meshed model, the class used for analysis.

```
(link-solid-geometry :class 'link-geometry-class
      )
(link-mesh-model :class 'link-mesh-class
      geometry-model-object ^link-solid-geometry
      mesh-database ^^mesh-database
      )
(analysis :class 'analysis-link-model-class
      mesh-model-object ^^link-mesh-model
```

```
            )
```

4.6 Meshing

In the context of meshing AML uses a methodology of geometry attribute tagging and tag propagation which allows resultant geometry from a boolean operation to refer back to the tags attached to the original geometry. This makes it possible to control mesh refinement on individual parts of the resultant geometry. All geometric classes that inherit from *tagging-object* will be tagged. The properties below are used to determine which entities of the geometry is to be tagged (points, edges, surfaces, solids) and attributes used in the context of meshing (maximum edge size, minimum edge size, curvature refinement, etc.).

```
tag-dimensions '(0 1 2 3)
tag-attributes '(0.25 0.0625 0 0.1 0 10.0 1e-5)
```

Meshing of the links is achieved through a single class, the *paver-mesh-class*. The assembled link geometry to be meshed is given to this object and it in turn creates the necessary information to pass to the meshing process.

```
(link-mesh :class 'paver-mesh-class
      object-to-mesh ^^geometry-model-object
      mesh-database-object ^^mesh-database
      element-shape :quadtri
      solid-mesh? nil
      )
```

After the mesh is generated, mesh entities can be queried using tagged-objects to retrieve nodes, elements, edges, faces, and regions from the mesh. By choosing a line as a cross section, the links are meshed as shell elements. By specifying *solid-mesh? t* and choosing a `:tet` element shape it should also be possible to generate a solid mesh, but this did not work.

To correctly simulate the boundary constraints between links, each mesh need to have a node in the joint center. The mating surface of a joint element is often not located at the joint center. A node is therefore created using *mesh-node-class*, which creates a node at the joint center.

```
(joint-center-node :class 'mesh-node-class
      coordinates (the origin (:from ^main-frame))
      mesh-object (the link-mesh (:from (the link-mesh-model (:from
      ^link-ref-incident-on-constraint))))
      )
```

The nodes associated with the mating surfaces are queried, using *mesh-nodes-query-class*, by referencing to the tagged object which constitutes the surface geometry.

```
(mating-surface-nodes :class 'mesh-nodes-query-class
      tagged-object-list (list ^^imprinted-pin)
      mesh-object (the link-mesh (:from (the link-mesh-model (:from
      ^link-ref-incident-on-constraint))))
      )
```

4.7 Analysis

The analysis can be performed using the integrated MSC Nastran software or external software. It has not been possible to find out if AML currently supports dynamic multibody analysis and it is instead more convenient to use an external application for the job.

FEDEM[5] developed at NTNU by Sivertsen, uses a finite element approach to dynamic simulation of mechanisms combining kinematics and structural analysis. FEDEM can import FE models in the MSC Nastran Bulk Data File (`.bdf`) and needs a Model file (`.fmm`) and a Solver input file (`.fsi`) in addition to run a simulation.

The Nastran Bulk Data File can be exported from AML. The *nastran-analysis-class* manages all communication between an instance of an *analysis-model-class* and the MSC Nastran software application. Demanding the run-nastran@ property will write the bulk data file (`.bdf`) with all node sets, element sets, property sets, load cases (boundary conditions), analysis types, and materials to the analysis directory and start the Nastran application.

```
(nastran-interface :class 'nastran-analysis-class
      analysis-model-object ^superior
      model-name (the folder (:from ^^mechanism-selection))
      nastran-file-name (concatenate (write-to-string  (object-name
      ^^superior)) ".bdf")
      nastran-version (nth 2 '(:nei-nastran :msc-nastran :nx-nastran))
)
```

The additional data for the Model file (`.fmm`) and a Solver input file (`.fsi`) data can be extracted from the AML model. A method was written for collecting the orientation matrices of frames located in joint positions and calculating the transformation matrices between them to create the output needed for the files.

[5] http://www.fedem.com/

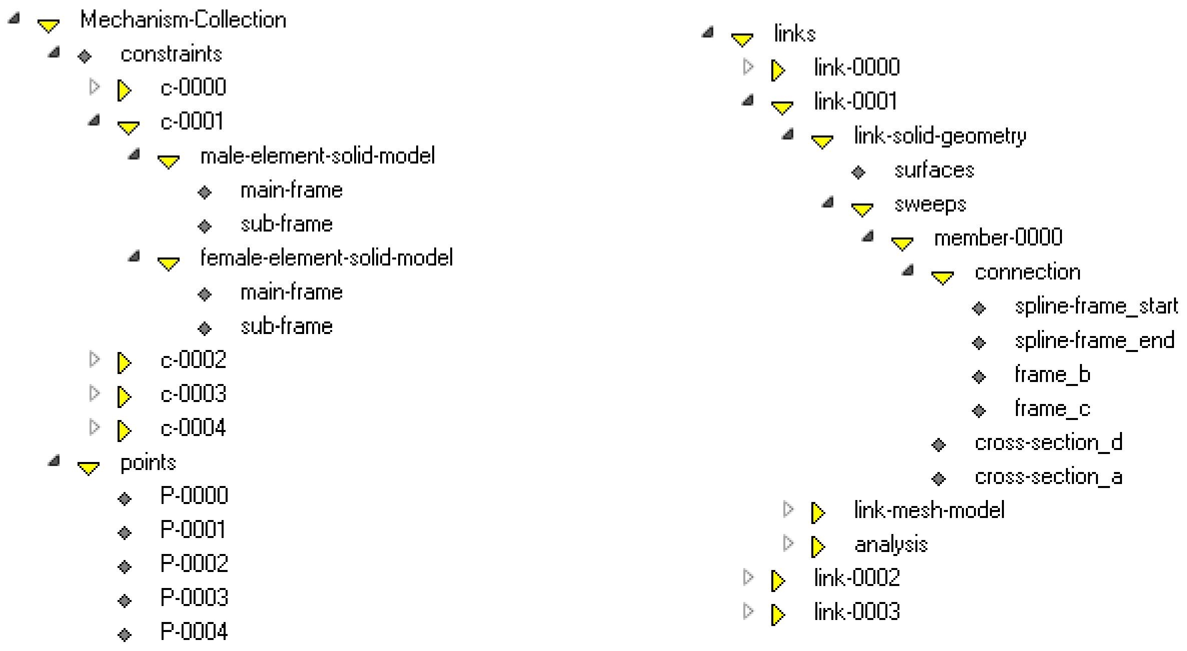
For the boundary conditions to be included in the bulk data file an *analysis-constraint-class* must be set in the *analysis-load-case-class*. For each link the nodes on the mating surfaces should be connected to the joint center with a MPC. In Nastran this is done using a RBE2 or RBE3. The *analysis-constraint-class* serves as a general class that is inherited into all other constraint classes. AML does not provide any documentation for other than *analysis-constraint-displacement-class*, but the classes *analysis-constraint-multi-point-type-1-between-1-to-many-nodes-class* and *analysis-rigid-body-element-type-1-class* seem fit for the job. Below is an example usage of the two classes.

```
(mpc :class 'analysis-constraint-multi-point-type-1-between-1-to-many-
nodes-class

      mesh-query-1 (the joint-center-node (:from ^^link-mesh-model))
      mesh-query-2 (the mating-surface-nodes (:from ^^link-mesh-model))
      )

(rbe :class 'analysis-rigid-body-element-type-1-class
      dependent-nodes-query-object (the mating-surface-nodes (:from
      ^^link-mesh-model))

      independent-node-query-object (the joint-center-node (:from
      ^^link-mesh-model))
      )
```

Either of them can be set in the *analysis-load-case-class* but running Nastran gives an error and it does not work. Support from TSI was unsuccessful and lead to a stop to the further development. The remaining work before the application is ready to export the mesh and create valid for simulation is to extract the node numbers connected to the joint center from the mesh and formatting this with the rest of the output.

5 Results

The goal was to make an application that works for as wide a variety of mechanism as possible. Let's have a closer look at how this works. Starting up the application, a few predefined mechanisms are available in a drop down menu. The four bar mechanism was selected and the model tree is shown



- Mechanism-Collection
  - constraints
    - c-0000
    - c-0001
      - male-element-solid-model
        - main-frame
        - sub-frame
      - female-element-solid-model
        - main-frame
        - sub-frame
    - c-0002
    - c-0003
    - c-0004
  - points
    - P-0000
    - P-0001
    - P-0002
    - P-0003
    - P-0004

- links
  - link-0000
  - link-0001
    - link-solid-geometry
      - surfaces
      - sweeps
        - member-0000
          - connection
            - spline-frame_start
            - spline-frame_end
            - frame_b
            - frame_c
          - cross-section_d
          - cross-section_a
    - link-mesh-model
    - analysis
  - link-0002
  - link-0003

Figure 5-1 Model tree in mechanism application

From the model tree all the parts of the mechanism are shown and it is possible to draw the wanted elements. The link geometry can be draw directly, but let's take a look at which role the different elements have in the application. Drawing all the point elements displays the joint positions given as input to the application, those are used for positioning the frames for each joint element

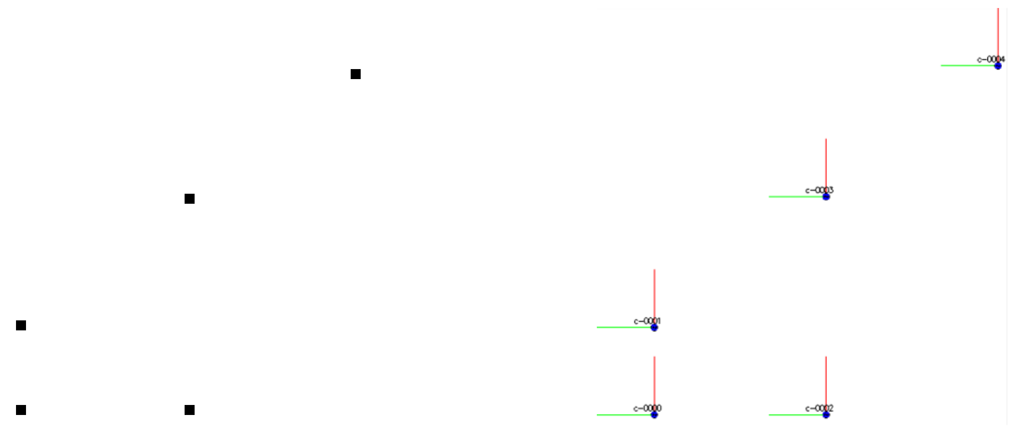


Figure 5-2 Joint positions and joint frames

Let have a closer look at link 1, connected between the two male joint elements of the joints furthest the left in Figure 5-2. The link is connected between the sub-frames of the joint elements (***z***-axis in blue), which is the starting point of the spline curve connecting the joints. As default for parallel joint axes, the curve generated is a straight line and the spline-frames are coincident with the sub-frames. If the shape of the curve is altered using design points, the ***x***-axes of the spline-frame are oriented tangentially to the curve. This is to make sure the cross section is oriented correctly.

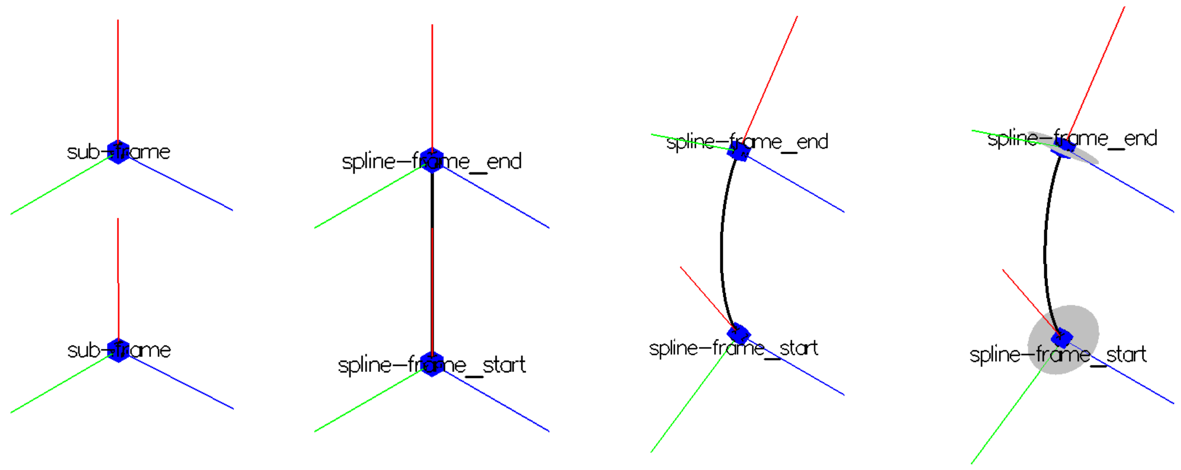


Figure 5-3 Building blocks of a link

The dimensions of the start and end cross section can be customized, as well as the cross section type. When the wanted shape, dimension and type is selected, the link geometry can be drawn, the possibilities are endless, some examples are shown in Figure 5-4.

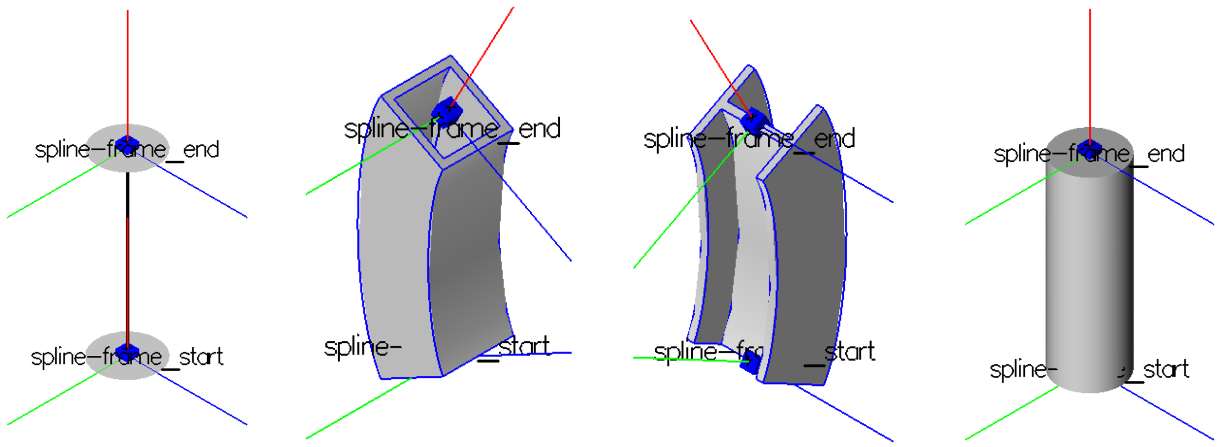


Figure 5-4 Examples of shape and cross sections of a link

From the dimensions of the link, the size of the joint is calculated. This is used to position the sub-frame relative to the joint center, where the main frame is placed. When the dimensions of the joint and link are both set, they are assembled together in the *link-solid-geometry* as seen in Figure 5-5.

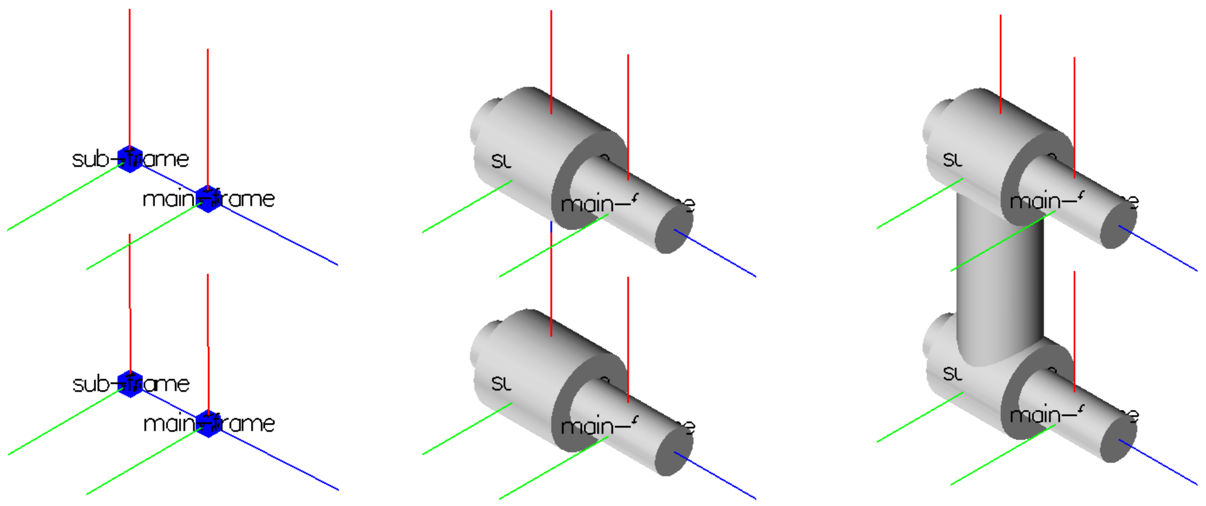


Figure 5-5 Positioning of joint and assembly of link

The different shape, dimensions and cross section types of a member of a link are characterized as morphological variations. The difference between a binary, ternary and quaternary link are the number of members on the link. Since all links are made from one generic class, the same morphological variations applies to links of higher degree as well. In Figure 5-6 is an example of a ternary link with female joint elements. Note that the sub-frame and main-frame of the joint are coincident.

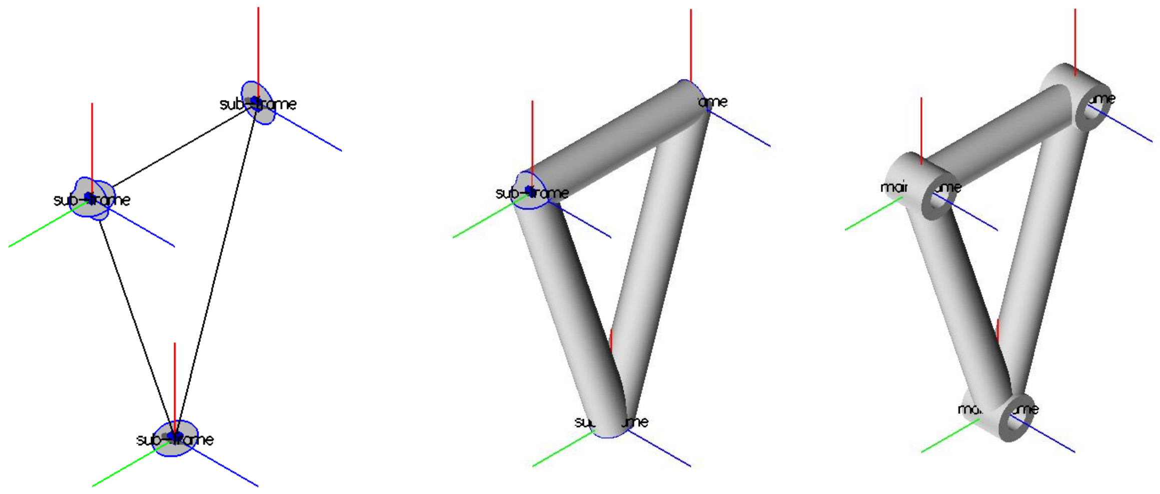


Figure 5-6 Assembly of ternary link

For quaternary links, the number of possible members is higher than needed to connect the joints. This means there are topological variations of the link. Figure 5-7 shows topological variations of a quaternary link.

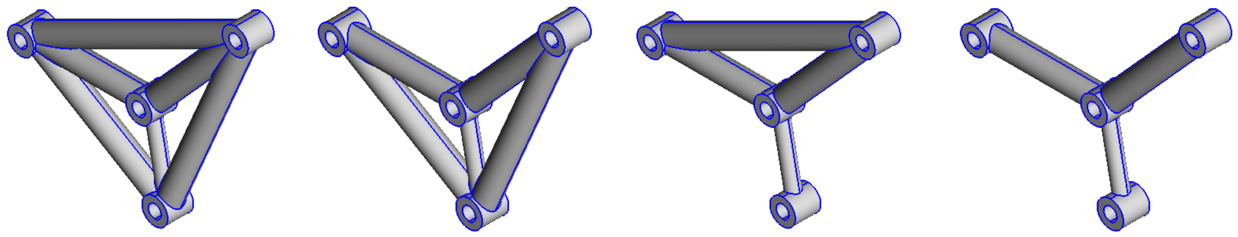


Figure 5-7 Different topological variations of a quaternary link

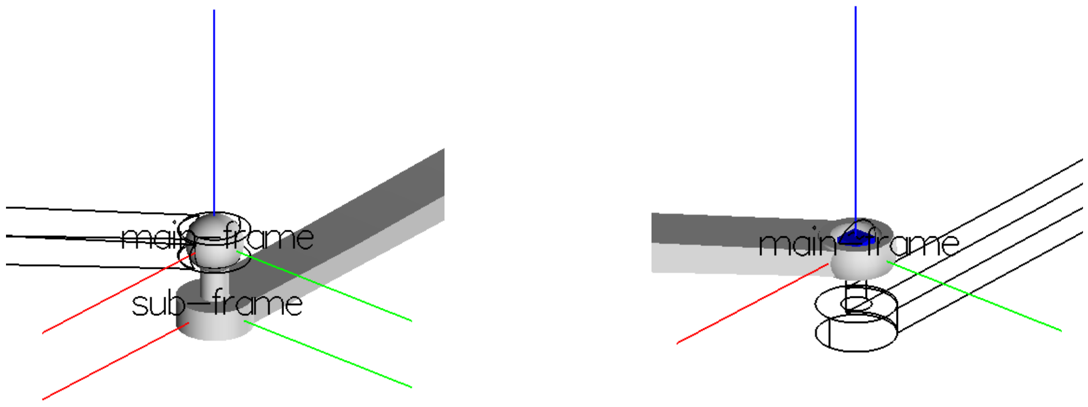


Figure 5-8 Position of sub-frame and main frame on sperical joint

The position of the sub-frame is calculated from the joint size and is dependent on the joint dimension, a spherical joint is seen in Figure 5-8. But in order for the joint elements to match, they have to be the same dimension. As seen from Figure 5-9, joint element size is dependent on the dimension of the connected cross section on both elements of a joint. The dimensions of the paired element might as well be controlling to the size of a joint.

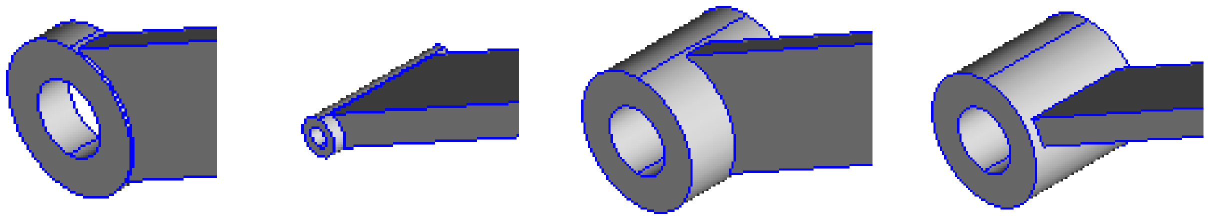


Figure 5-9 Joint element size varying with cross section on same link and paired element.

After a link is modelled it can be meshed. This is done in using from the *link-mesh-model* in the model tree. If a line is selected as cross section, the mesh will be a shell element model.

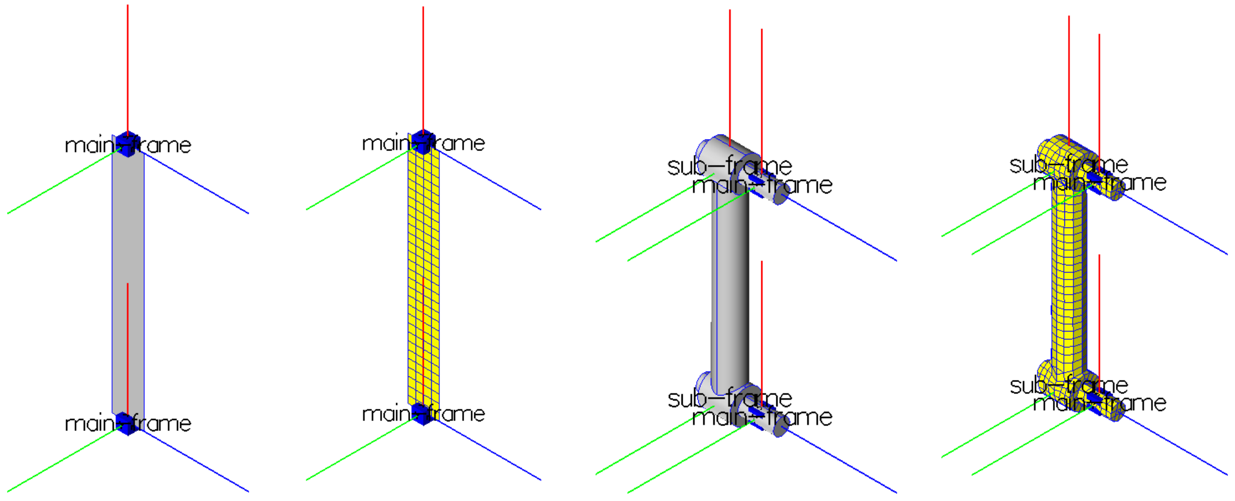


Figure 5-10 Meshing of links with different cross sections

From the mesh, nodes or elements from a tagged part can be queried. The nodes from the mating surfaces of the joint elements and a node in the joint center are needed to create the MPC, from Figure 5-11 shows the queried nodes from a revolute joint and a spherical joint. But as mentioned it was not possible to make a MPC between them in the analysis

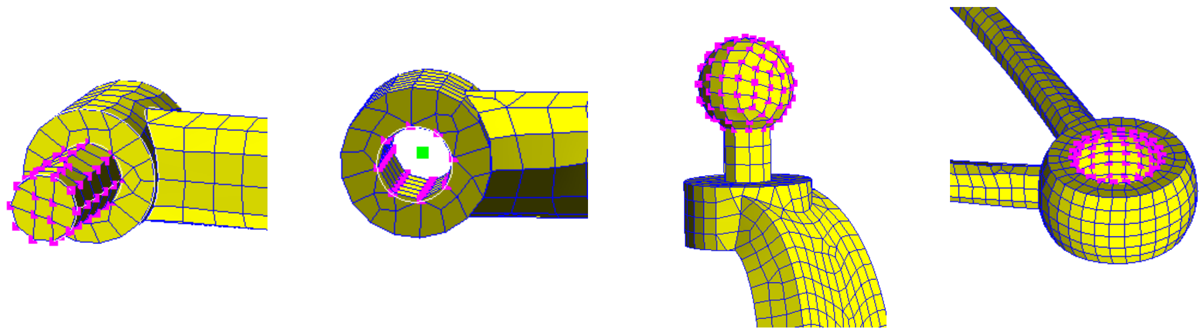


Figure 5-11 Nodes on mating surfaces of joint elements and node in joint center

The links and joints are the basic building block of the mechanism. A major importance is that the application correctly assembles them according to the given topology. Figure 5-12 shows and example of a four bar mechanism generated with circular and line cross section on the links and a Watt mechanism with circular cross section links. For any mechanism, when the points defining the joint positions are changed, the model will update automatically. The same applies to changing the shape, dimension and type of cross section or visible members on a link. If changing the topology of a mechanism, e.g. removing or adding a joint, the instance tree is updated, and the whole model is redrawn.

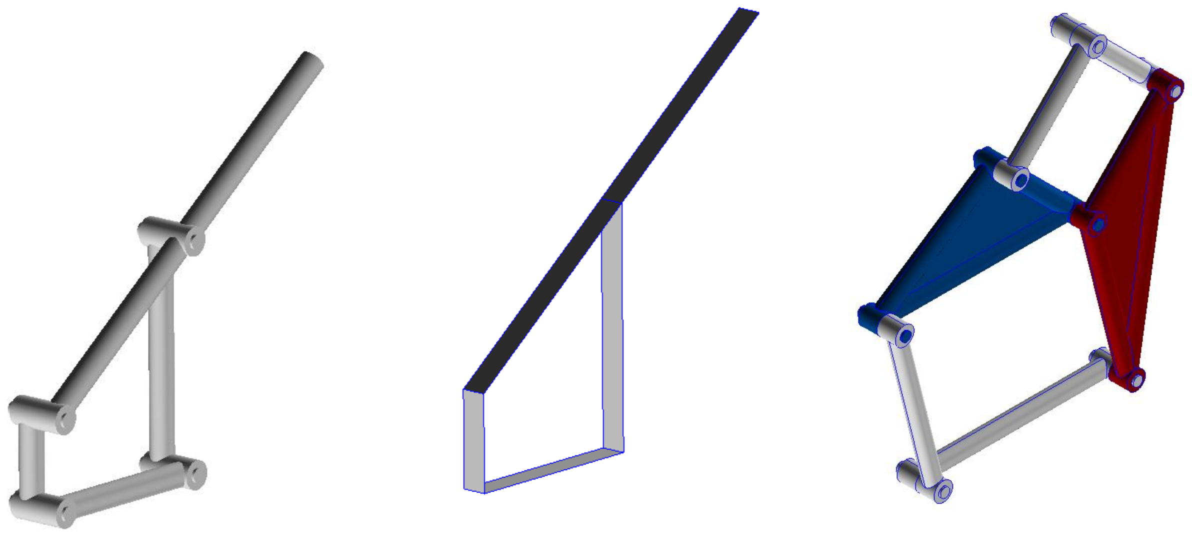


Figure 5-12 Four bar mechanism and Watt mechanism generated by the application

From the model the mesh for the wanted links can be generated. If any changes are made to the model, the mesh is automatically updated to match the new geometry. Figure 5-13 shows the mesh generated for the models shown in Figure 5-12.

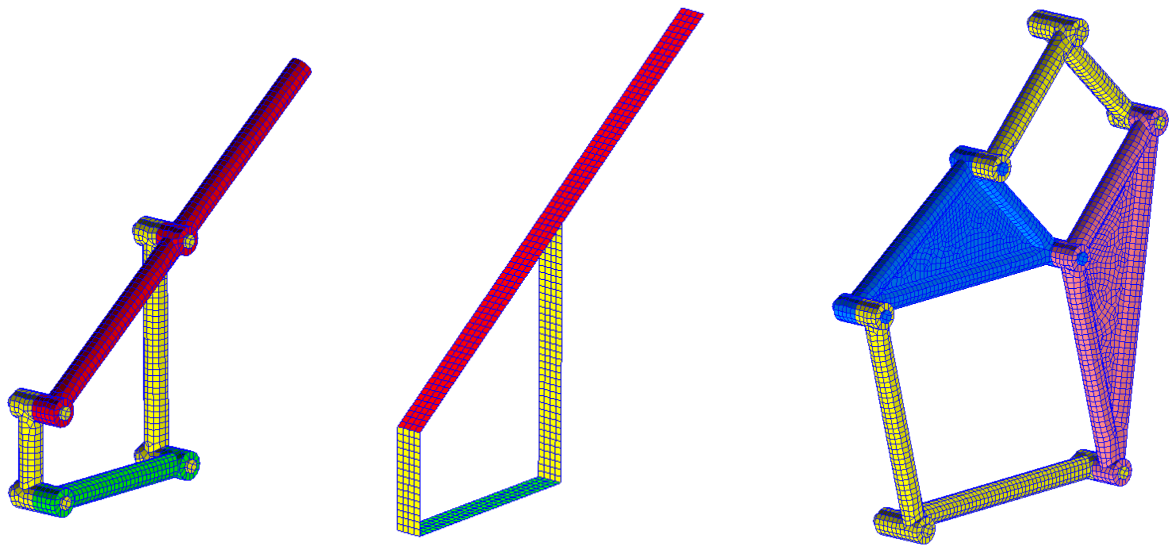


Figure 5-13 Mesh generated by the application of models in Figure 5-12

When the mesh is ready the analysis class can be run and Nastran bulk file generated. The `.bdf` file contains overview over the elements and nodes of the mesh connected to a link.

To test out a more challenging shape of a link, the lower control arm of a car suspension was modelled. The shapes of the members were controlled using design points and the link was successfully meshed as seen in Figure 5-14.

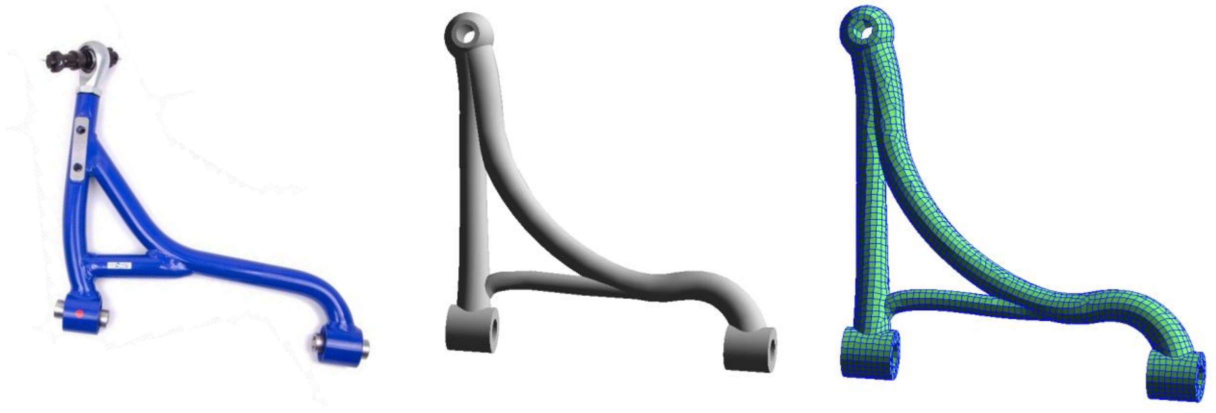
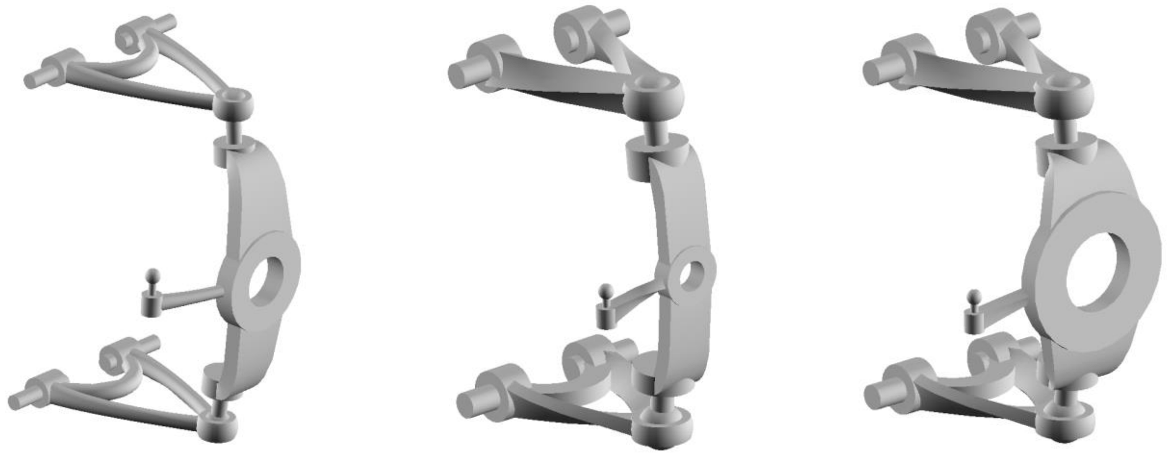


Figure 5-14 Lower control arm from car suspension

A little more interesting example is to model a double wishbone car suspension. An early proof of concept was implemented in the program, created using the same input format as with the other examples. Some additional design points have been specified for the members where the automatic curve generation didn't give the wanted shape. Below are a few input variations of the suspension mechanism, the different parts are of course possible to mesh as well, as shown in Figure 5-15.



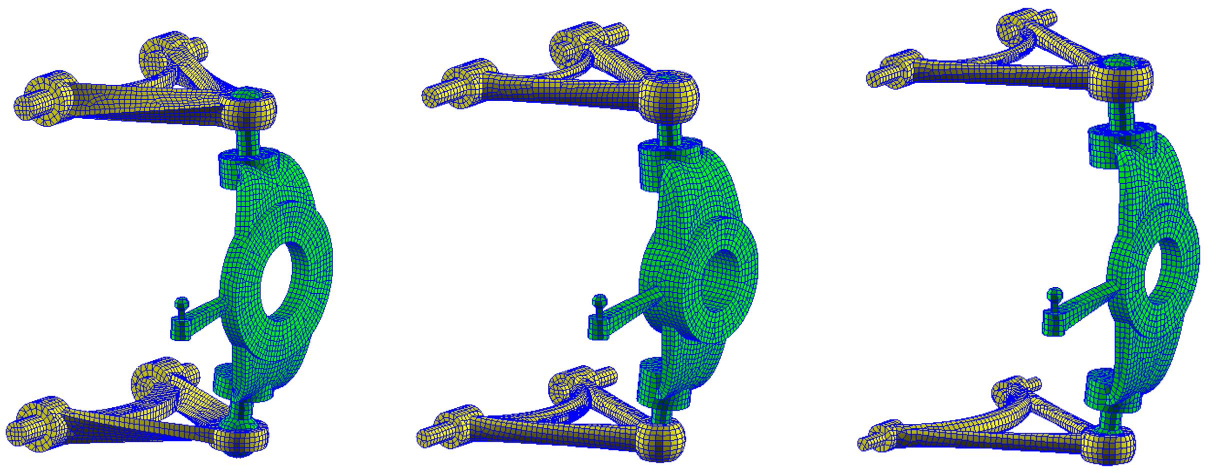


Figure 5-15 Double wishbone suspension showing different variations of models and mesh

6 Discussion

During the development, code has presented and explained in order to argument for the design and implementation choices. The results shown are promising, but hard to evaluate against any performance criteria. The different research questions stated are revisited and discussed here.

> **RQ1:** How can Sheth – Uicker convention be utilized to automate the generation of link geometry?

The parameters provide an efficient way to describe the link displacements in a mechanism. When the link's size is fixed its parameters are unrelated to position and orientation. This is useful when calculating the displacement from a new joint position or orientation, but requires joint variables to change accordingly in order to not create any inconsistencies of the assembly.

In the application the joint position and orientation are parameterized, this is much easier and more intuitive than using the SU parameters as input to the application. This mean the link size is not fixed and subject to change given by the joint input. If one were to use the SU parameters as input it would be necessary to reassemble the model whenever a parameter changes. The SU parameters are mainly considered as analysis tool and are not as useful when designing a mechanism model.

What makes the SU parameters more interesting is to use them to generate the shape of a link. This creates a parametrized shape relative to the joints that is still valid when the position or orientation of the joint changes. The SU convention is not the only way to achieve shape parameterization, but because of its importance in mechanism theory it was natural to study it as a part of this thesis. As we have seen this works for planar and spherical mechanism, but with an incident or skew line configuration there isn't one general shape that works as good. An option may be to make a library of parameterized shapes which then are selected for a member.

> **RQ2:** How can different links be represented using one generic class?

Identifying and utilizing morphological and topological variations of links shows that it is possible to make a generic class that can model an unlimited variety of links regardless of degree, shape and dimension. This approach creates a flexible and robust model. Functionality for adding surfaces between closed curves is presented but only works for three curves and not in every case. Another implementation using parameterized surface should be considered.

Currently the custom design points are given using global coordinates. This may lead to problems if the joint positions move, causing an unwanted shape of the link. A better approach would be to define position relative to the joints e.g. by using the shape parameters.

> **RQ3:** How will the physical dimensions of links and joints affect each other?

In the initial stages of development the application closely followed the concepts of the SU convention. This lead to problems when the curves used for generating the shape did not start in the same position as the frames used in the SU convention. This displacement was caused by the physical dimensions of a joint and was solved by adding a "construction" frame. Still preserving the parameters of SU when no physical joint is present, (as seen in the case with shell elements), the frame adds a physical dimension to the joint displacement where there before only was relative motion described by joints variables.

The position of the construction frame is unknown until the joint dimensions are known. This was solved by relating joint size to link size, it could have been done the other way around, depending on what's the critical components of the system.

The frame also gives more flexibility for choosing which part of a joint is considered to be the joint center. This is convenient when positioning the frames related by the joint variables and the node specifying the connection to the surrounding mating surfaces by a multi-point constraint.

> **RQ4:** How can mesh generation be automated to create a link between modeling and simulation?

The meshing functionality in AML and the procedure presented shows that it is possible to automate the modeling and meshing process. Not everything worked as hoped, but AML has a lot of functionality for mesh generation implemented. The main problem is that it is poorly documented and TSI was not able to give any support.

Before simulation, Nastran had to be run in order for the mesh to be exported as a `.bdf` file. This is not very efficient as Nastran tries to run an analysis from the data given in AML. The model is already meshed and it would be better to export the mesh information directly from AML. It was not necessary for demonstration of this thesis, but regarding future work it should be considered to either write custom functions to export mesh information or run the entire analysis in AML with Nastran. It is not certain if AML currently supports dynamic multibody analysis and it may not be as efficient as the FEDEM software.

Other thoughts

In contrast to other popular programming languages, AML does not have an online community and few internet resources exist. A Google Scholar search of “aml technosoft” or “adaptive modeling language technosoft” gives 180 results, with only a handful of code examples. If something doesn’t work, there is usually no way to find the answer. It would also be worth putting some extra effort into using AML in a more modern editor. XEmacs is considered outdated, and a more extensible and customizable editor such as Sublime Text 2 would increase productivity and lower to threshold for getting started with AML.

7 Conclusions

This thesis explores to possibilities of using the Sheth – Uicker convention together with Knowledge Based Engineering for developing an application used in design automation of mechanisms. The concepts have been implemented and tested using the TechnoSoft Inc. supported Adaptive Modeling Language.

A proposal for including an extra frame in addition to the SU convention gives an extra reference for describing the joints displacement caused by the solid geometry of a joint element and creates a basis for generating the shape of a link.

The SU parameters do not prove to be significant as an input format or in designing the mechanism model but show promising results as a basis for generating the shape of links. It is not a universal technique, but also allows the shape to be customized with additional design points.

A generic class that can model an unlimited variety of links regardless of degree, shape and dimension is developed showing the flexibility and robustness of using concepts from KBE in modelling.

The application is the start of a solid framework for developing a fully automated design process of mechanisms. The application successfully shows automated modelling and meshing of mechanisms from given input parameters.

8 Further work

The following section suggests possible areas for further development of the application.

The application currently only supports reading input from file. A better GUI for editing mechanism specifications in the application should be made, also allowing saving an edited model to file. Refining the input format may be necessary and possibly look at database integration with AML. Other formats may be easier to integrate with other applications. The input format should also handle parameters for handling mesh refinement in critical areas detect in simulation.

The SU parameters are a powerful way of representing the link and joint displacements. It would be interesting to see if these parameters could be even more tightly integrated with the application especially regarding mechanism synthesis and kinematic analysis.

Examples of lower pair joints are shown in the application, more joint types and also higher pairs should be implemented.

The initial dimension of links has a default value unless input is given. Using classical beam theory and solving the mechanism as a truss system can give a good initial estimation of required link dimension. This will reduce the work done by an optimization loop.

The mesh size is currently set manually, calculating an initial mesh size depending on part size should be implemented and also identifying and applying mesh refinement in critical areas.

Multi-Point Constraints do not currently work in the application. Looking deeper into this and ways of exporting mesh data generated in AML both with and without running Nastran should be considered. It might also be possible to run the full analysis in AML without the need for external software. This should be looked further into.

The results acquired from an analysis should be interpreted and compared against objective functions. This should generate new input to the application and make it possible to run an optimization loop.

References

Amadori, K., Tarkian, M., Ölvander, J., & Krus, P. (2012). Flexible and robust CAD models for design automation. doi: 10.1016/j.aei.2012.01.004

Bongardt, B. (2013). Sheth-Uicker convention revisited. *Mechanism and Machine Theory, 69*, 200-229. doi: DOI 10.1016/j.mechmachtheory.2013.05.008

Chemaly, A. (2006). Adaptive Modeling Language and Its Derivatives. *NASA Tech Briefs*.

Elstad, T. A., & Granlund, S. H. (2013). *Integrating general-purpose software design tools into KBE-development.* (Project thesis), Norwegian University of Technology and Science, Trondheim.

Felippa, C. A. (2004). *Introduction to finite element methods*: Department of Aerospace Engineeing Sciences, University of Colorado Boulder.

Pinfold, M., & Chapman, C. (1999). Design engineering – a need to rethink the solution using knowledge based engineering.

Pinfold, M., & Chapman, C. (2001). The application of a knowledge based engineering approach to the rapid design and analysis of an automotive structure.

Rocca, G. L. (2012). Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design. doi: 10.1016/j.aei.2012.02.002

Roth, S., Chamoret, D., Badin, J., & Gomes, S. (2011). Crash FE Simulation in the Design Process - Theory and Application. *Numerical Analysis - Theory and Application*. doi: 10.5772/23813

Sheth, P. N. (1972). *A Digital Computer Based Simulation Procedure for Multiple Degree of Freedom Mechanical*

Systems with Geometric Constraints. University of Wisconsin, Madison, WI: University Microfilms, Inc., Ann Arbor, MI.

Sheth, P. N., & Uicker, J. J. (1971). A Generalized Symbolic Notation for Mechanisms. *Journal of Engineering for Industry, ASME Transactions, 93*, 102-112.

Sivertsen, O. I. (2001). *Virtual testing of mechanical systems, theories and techniques*: Swets & Zeitlinger B.V., Lisse.

Sivertsen, O. I. (2014). Proposed Approach for Introducing Automation in Mechanism Design.

Skaare, R. K. (2014). *Introducing automation in design of mechanism.* (Project thesis), Norwegian University of Science and Technology, Trondheim.

TechnoSoft. (2010). AML Reference Manual 5.0B5. *TechnoSoft Inc.*

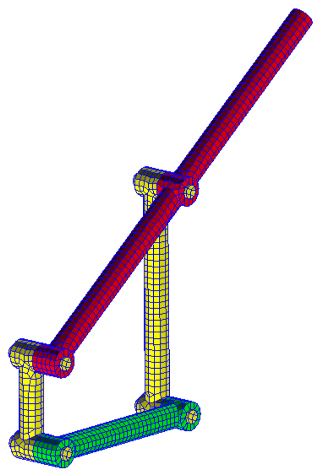
TechnoSoft. (2012). *AML Basic Training Manual V3.06*: TechnoSoft Inc.

Uicker, J. J., Ravani, B., & Sheth, P. N. (2013). *Matrix Methods in the Design Analysis of Mechanisms and Multibody Systems*: Cambridge University Press.

Appendix A Additional results

A.1 Four bar mechanism

Here are the input files used for generating the four bar mechanisms with circular and shell cross section. The coordinates.txt and constraints.txt are the same, by changing from “line” to “circular” in shapes.txt the mechanism changes.



constraints.txt

```
0     "revolute"  "(1 0)"     "(0 0 1)"   "()"
1     "revolute"  "(1 2)"     "(0 0 1)"   "()"
2     "revolute"  "(3 0)"     "(0 0 1)"   "()"
3     "revolute"  "(3 2)"     "(0 0 1)"   "()"
4     "open"      "(0 2)"     "(0 0 1)"   "()"
```

coordinates.txt

```
0.0   0.0   0.0
0.0   0.15  0.0
0.3   0.0   0.0
0.3   0.375 0.0
0.6   0.6   0.0
```

shapes.txt

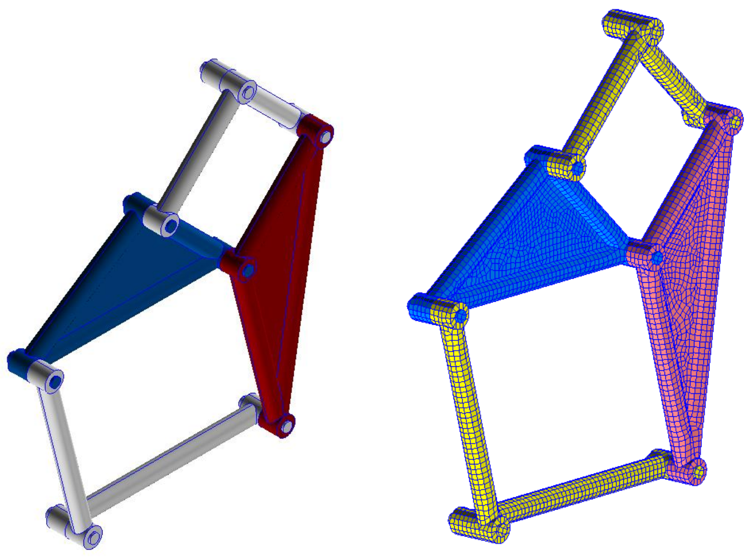
```
2     1     "nil" "(0.02 0.02)"     "()"
default     default     "line"      "(0.02 0.02)"     "()"
```

shapes.txt

```
2       1       "nil" "(0.02 0.02)"     "()"
default     default     "circular"  "(0.02 0.02)"     "()"
```

A.2 Watt-1 mechanism

Here are the input files used for generating the six bar mechanisms, also known as watt mechanism.



constraints.txt

```
0     "revolute"  "(0 1)"     "(0 0 1)"   "()"
1     "revolute"  "(0 2)"     "(0 0 1)"   "()"
2     "revolute"  "(3 1)"     "(0 0 1)"   "()"
3     "revolute"  "(3 4)"     "(0 0 1)"   "()"
4     "revolute"  "(3 2)"     "(0 0 1)"   "()"
5     "revolute"  "(5 2)"     "(0 0 1)"   "()"
6     "revolute"  "(5 4)"     "(0 0 1)"   "()"
```

coordinates.txt

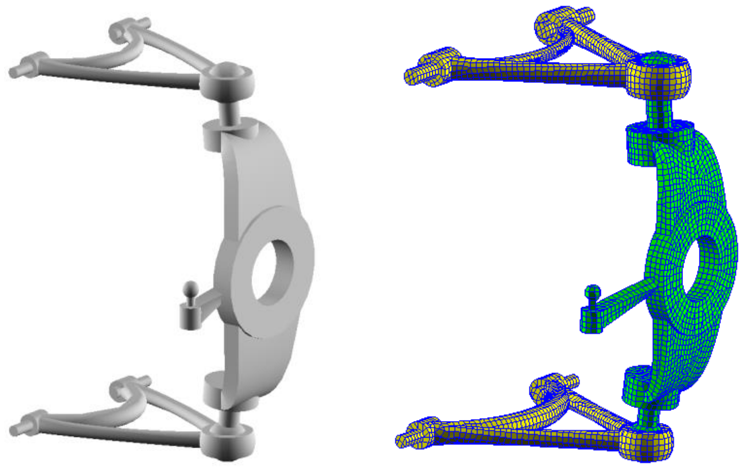
```
0.0   0.0   0.0
0.5   0.0   0.0
-0.1  0.4   0.0
0.2   0.6   0.0
0.4   0.4   0.0
0.6   0.6   0.0
0.4   0.8   0.0
0.3   0.9   0.0
0.7   0.7   0.0
```

shapes.txt

```
default     default     "circular " "(0.04 0.04)"     "()"
```

A.3 Double wishbone suspension

Here are the input files used for generating the double wishbone suspension.



constraints.txt

```
0     "revolute"  "(0 1)"     "(-1 0 0)"  "()"
1     "spheric"   "(1 2)"     "(0.1 0 1)" "((0 0 1))"
2     "spheric"   "(1 3)"     "(0.1 0 -1)"      "((0 0 -1))"
3     "spheric"   "(1 4)"     "(0 0 1)"   "((0 0 1))"
4     "revolute"  "(2 5)"     "(0 1 0)"   "()"
5     "revolute"  "(2 5)"     "(0 -1 0)"  "()"
6     "revolute"  "(3 6)"     "(0 1 0)"   "()"
7     "revolute"  "(3 6)"     "(0 -1 0)"  "()"
8     "open"      "(0 4)"     "(0 0 1)"   "()"
9     "open"      "(0 5)"     "(0 0 1)"   "()"
10    "open"      "(0 6)"     "(0 0 1)"   "()"
```

coordinates.txt

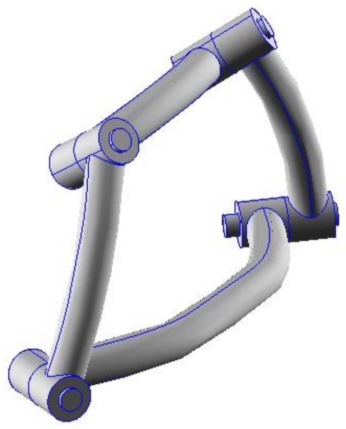
```
0.0   0.0   0.0
0.25  0.0   1.0
0.25  0.0   -1.0
0.1   0.5   0.0
1.25  0.5   1.0
1.25  -0.5  1.0
1.25  0.5   -1.0
1.25  -0.5  -1.0
1.75  0.25  0.0
1.5   0.0   1.0
1.5   0.0   -1.0
```

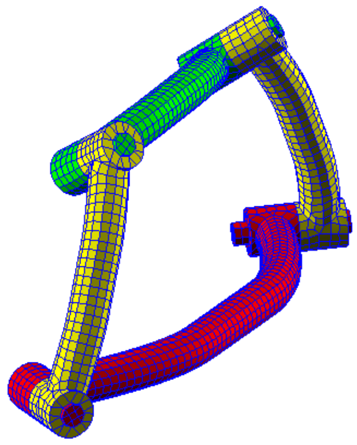
```
0.0    0.0    0.8
0.0    0.0    -0.8
1.0    -0.5   1.0
1.0    0.5    1.0
1.0    -0.5   -1.0
1.0    0.5    -1.0
0.0    0.3    0.0
```

shapes.txt

```
3      0      "rectangle" "(0.1 0.05)"      "(16)"      "(0.5)"
3      1      "rectangle" "(0.1 0.05)"      "(15)"      "(0.5)"
3      2      "rectangle" "(0.1 0.05)"      "(2)" "(0.4)"
2      0      "rectangle" "(0.1 0.05)"      "(14)"      "(0.5)"
2      1      "rectangle" "(0.1 0.05)"      "(13)"      "(0.5)"
2      2      "rectangle" "(0.1 0.05)"      "(1)" "(0.4)"
2      default     "rectangle" "(0.1 0.05)"      "()"
1      0      "rectangle" "(0.6 0.1 0.2 0.1)"     "(11)"
1      1      "rectangle" "(0.6 0.1 0.2 0.1)"     "(12)"
1      2      "rectangle" "(0.1 0.1)" "(17)"
1      3      "nil" "(0.1 0.1)" "()"
1      4      "nil" "(0.1 0.1)" "()"
1      5      "nil" "(0.1 0.1)" "()"
1      default     "circular"  "(0.1 0.1)" "()"
default     default     "circular"  "(0.04 0.04)"     "()"
```

A.4 Spherical four bar mechanism





constraints.txt

```
0      "revolute"  "(1 0)"      "(0.7071 -0.7071 0.0)"  "()"
```

```
1     "revolute"  "(1 2)"     "(0.866 0.5 0.0)" "()"
2     "revolute"  "(3 0)"     "(0.75 -0.433 0.5)"     "()"
3     "revolute"  "(3 2)"     "(0.813797 0.296198 0.5)"     "()"
```

coordinates.txt

```
0.7071      -0.7071     0.0
0.866 0.5   0.0
0.75  -0.433      0.5
0.813797    0.296198    0.5
```

shapes.txt

```
default     default     "circular"  "(0.08 0.08)"     "()"
```

A.5 Additional images

To demonstrate some of the capabilities of the application some screenshots have been taken which show the output generated by varying the input parameters. In Figure 0-1, different types of cross section and dimensions are demonstrated and in Figure 0-2 the shapes have been tweaked using design points.

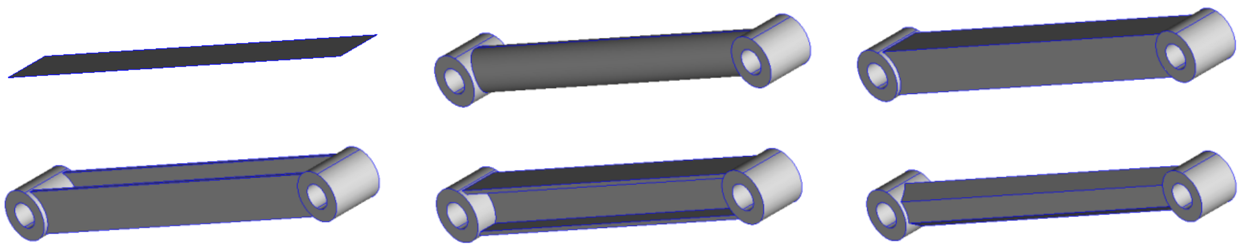


Figure 0-1 Different cross sections and dimensions of a binary link

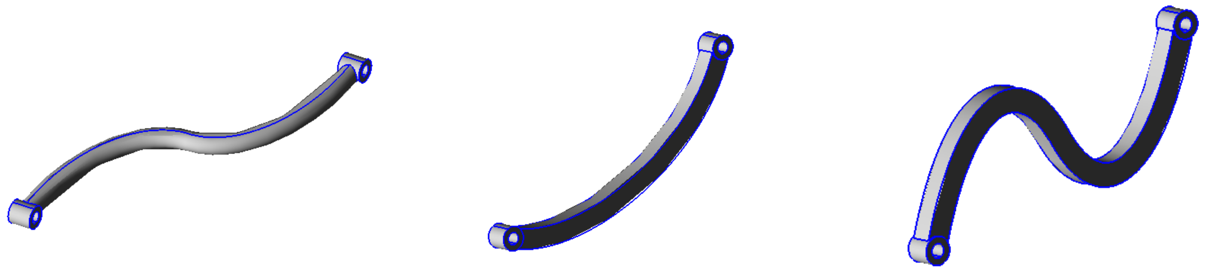


Figure 0-2 Different shapes of a binary link

Since all links are defined from one common class the same customization is also possible for ternary links. Figure 0-3 shows different morphological variations of a ternary link.

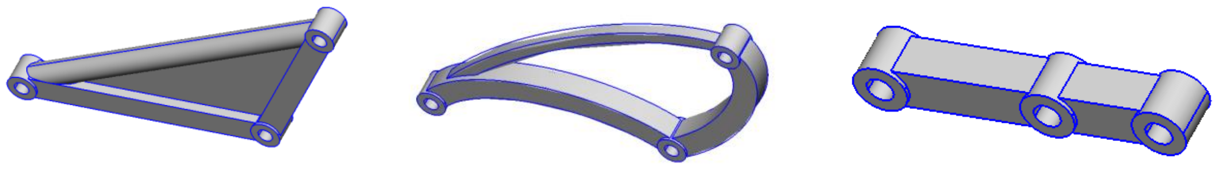
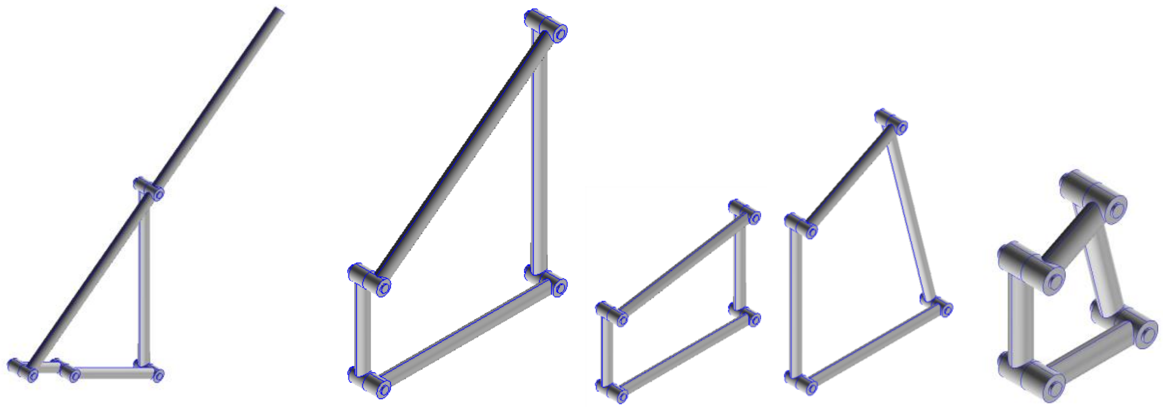


Figure 0-3 Different morphological variations of a ternary link

Here are some additional images of mechanism generated using the application.



Appendix B AML Reference Manual

Below are commonly used functions and classes, some of them covered in the AML Reference Manual (TechnoSoft, 2010), others not.

B.1 Functions

apropos [FUNCTION]

Arguments: (symbol)

Optional arguments: (package)

concatenate [FUNCTION]

Remaining arguments: (strings)

Example *concatenate*:

```
> (concatenate "a" "b" "c")
;; returns: "abc"
```

compile-system [FUNCTION]

Arguments: (system)

Keyword arguments: (:force? :forget? :new? :all? :compiled?)

describe [FUNCTION]

Arguments: (object)

logical-path [FUNCTION]

Arguments: (pathname)

Remaining arguments: (directories/file-name)

Example *logical-path*:

```
> (logical-path :aml "modules")
"C:\Program Files\Technosoft\AML\AML5.85_x64\modules"
```

make-sequence [FUNCTION]

Arguments: (type size)

Keyword arguments: (:initial-element)

Example *make-sequence*:

```
> (make-sequence 'array 5)
;; returns: (nil nil nil nil nil)
> (make-sequence 'string 3 :initial-element '#\a)
;; returns: "aaa"
```

map [FUNCTION]

Arguments: (type function list)

Remaining arguments: (lists)

Example *map*:

```
> (map 'fixnum '1+ '(1 2 3 4) )
;; returns: (2 3 4 5)
> (map 'fixnum '- '(1 2 3 4) '(8 8 8 8))
;; returns:(-7 -6 -5 -4)
```

methods-for-class [FUNCTION]

Arguments: (class-or-instance)

position [FUNCTION]

Arguments: (item sequence)

Keyword arguments: (:key :from-end :test :test-not :start end)

read-from-string [FUNCTION]

Arguments: (string)

Optional arguments: (eof-errorp eof-value)

Keyword arguments: (:start :end :values?)

Example *read-from-string*:

```
> (read-from-string "string")
;; returns: string
> (read-from-string "4")
;; returns: 4
> (read-from-string "(list 1 2 0)")
;; returns: (list 1 2 0)
> (read-from-string "(1 0 0)")
;; returns: (1 0 0)
```

*** [FUNCTION]**

Optional arguments: ((number1 (default 1)) (number2 (default 1)))

Remaining arguments: (numbers)

Example ##. * example

```
> (*)
;; returns: 1
> (* 2)
;; returns: 2
> (* 2 3)
;; returns: 6
> (* 2 3 4)
;; returns: 24
```

B.2 Classes

analysis-constraint-class [CLASS]

This class serves as a general class that is inherited into all other constraint classes.

analysis-load-case-class [CLASS]

This class manages the boundary conditions of a part in a finite element analysis. A load case is a combination of a set of loads and a set of constraints.

Properties:

- constraint-objects-list
- load-objects-list

analysis-model-class [CLASS]

This is the base class which manages communication and all interfaces with the analysis application. Analysis types available, :buckling, :modal, :linear-static :direct-complex-eigenvalues, :modal-complex-eigenvalues, :static-aeroelastic-response, :multi, :flutter, :mfreq, :nonlinear-static, :nonlinear-implicit.

Properties:

- analysis-type
- mesh-object
- material-catalog-object

- load-case-objects-list
- materials-list
- property-set-objects-list
- element-set-3d-objects-list
- element-set-2d-objects-list
- element-set-1d-objects-list
- node-set-objects-list

analysis-node-set-class [CLASS]

This class is used to specify which nodes will eventually be used in an analysis by setting the node-set-objects-list property on an instance of the *analysis-model-class*.

Properties:

- query-objects-list

difference-object [CLASS]

The difference-object will create a single geometric instance consisting of only the regions of two objects that are not in common. If the first object in the property object-list is a NULL geom, then the result is a NULL geom.

Inheritance: boolean-object

Properties:

- simplify?

general-sweep-class [CLASS]

This class creates a sheet or solid body by sweeping a set of wire or sheet profiles along a curve path.

Properties:

- profile-objects-list
- path-points-coords-list
- profile-match-point-coords-list
- simplify?
- path-object
- tangential-sweep?

geometry-with-split-periodic-faces-class [CLASS]

This class creates geometry generated by imprinting a sheet-object instance into a symmetrical part of the source-object and results in an intersection curve embedded in surface of the source-object

Properties:

- source-object

nastran-analysis-class [CLASS]

This class manages all communication between an instance of an analysis-model-class and the MSC Nastran software application, enables the writing of a bulk data file (deck), and enables the running of Nastran.

Properties:

- analysis-model-object
- analysis-directory
- nastran-file-name
- data-file
- run-nastran@

nurb-curve-object [CLASS]

Inheritance: nurb-object, curve-object

Properties:

- degree
- points
- knots
- rational?
- homogeneous?

Example *nurb-curve-object*:

```
(define-class nurb-curve-test
      :inherit-from (nurb-curve-object)
      :properties (
            points '(   (0 0 0)
                        (1 -1 0)
                        (2 1 0)
                        (3 2 0)
                        (4 1 0)
                        (5 0 0)
                        (6 -2 0)
```

```
                                    (7 0 0))
                        degree 2
                )
        )
```

series-object [CLASS]

This class creates an arbitrary number of subobjects. The classes, quantity and properties of the subobjects created are based on the property formulas of class-expression, quantity, and init-form provided. Each subobject created has three properties called index, series-previous and series-next automatically added to it.

Properties:

- series-prefix
- init-form
- quantity
- class-expression

surface-from-three-edge-curves-class [CLASS]

This class generates a surface patch from three connected curve objects. The three curves must form a closed loop. The geometry of each curve object must be a single curve.

Properties:

- edge-1-object
- edge-2-object
- edge-3-object

B.3 Methods

vertex-of-object [METHOD]

Defined on Classes: graphic-object

Arguments: (instance)

Optional arguments: (vertex-id)

convert-coords [METHOD]

Defined on Classes: position-object

Arguments: (instance point)

Keyword Arguments: (:from :to)

B.4 Macros

default [MACRO]

When specified as the formula for a property, default will look up the tree for a property with the same name. If one is found, the value of that property is used. Otherwise, the specified default-formula is used.

Optional Arguments: (default-formula)

define-class [MACRO]

Define is used to define a new AML class. Defining a new class allows instances of that class to be created.

Arguments: (name)

Keyword arguments: (:inherit-from :properties :subobjects)

define-system [MACRO]

The main mechanism for defining systems. The definition should appear in a system definition file named system.def.

Arguments: (system-name)

Keyword arguments: (:require-libs :require-systems :require-loaded-systems :files :require-resources)

the [MACRO]

the is used to reference an object or a property in an AML object tree. The default behavior is as follows: A reference to an instance that inherits from property-class will return the value of that property. A reference to any other object will return the instance of the object.

Optional Arguments: (reference-path)

Keyword Arguments: (:from :eval? :dependencies?)

Appendix C AML setup

The AML Reference Manual (TechnoSoft, 2010) gives a short introduction to the core system components of AML, here is a short summary, with a supplementary explanation of the interpreter and editor integration and examples of the `aml-init.tsi` and `logical.pth`-files used in this thesis.

C.1 Interpreter

The AML process can be initiated directly from the command line. To start AML in the console, a minimal environment has to be set up using a batch file or running the following statement from the command line in the AML working directory.

```
> AMLConsole.exe AML.exe AML.img
```

The AML process is launched from AML.exe with the AML.img. This gives us a running AML interpreter in the command line. Interacting with the interpreter is then as simple as writing commands in the command line, and executing them with return. Errors also give rise to a pop-up dialog, where the user can choose to debug or abort. Choosing the debug option prints the stack trace to the console.

C.2 Editors

Text editors can use an REPL (Read Eval Print Loop) application to control the AML process and implement the full functionality of the AML interpreter. Currently XEmacs is the only editor supported by AML. But Elstad and Granlund (2013) makes an AML REPL for Sublime Text 2, unfortunately this does not work with AML 5.85.0. Oluf Tonning (2013) used the approach to successfully integrate the AML development framework into the Eclipse IDE.

The main benefit of integrating AML into Sublime is to give the developer several options to choose from. Some of the features include a modern default key-binding, familiar from most other applications. One example is Ctrl-c for copy and Ctrl-v for paste. Sublime has a more modern GUI, with a tab system that closely matches modern tabbed applications, like web browsers. Browsing and editing the project files and folders can easily be done in the left side bar of the editor. Sublime has a large and active plug-in community. This gives the developer access to , easily installed to the Sublime package manager. A mini-map on the right side of the editor, gives the developer an instant overview of the open file.

C.3 Images

An image is the main underlying file from which AML starts. An image holds all the AML classes, methods and functions definitions. The user can create a new AML "image" file that includes user defined classes, methods and functions. This image file can replace the default `.img` file the current AML version is using. Saving an AML image file should be done before calling the function *aml*.

C.4 Packages

Packages allow the user to define code in multiple name-spaces. Typically, most development will be done in the `:aml` package. Therefore, the user needs to state *(in-package :aml)* at the beginning of all AML source code files because you will be using functionality from the `:aml` package. When using functionality between packages, you must specify the package name as a prefix.

C.5 Modules

This function compacts a collection of AML systems and their resources into one module. An AML module consists of a directory containing one `.btc` file (that includes all `.btc` files from all required AML systems) and other resource directories based on the "require-libs" and "require-resources" keywords of the included AML systems. Modules are loaded into AML using the function *load-module*. A directory named "module-name-version" will be created by create-module.

C.6 Patches

A software patch is a file that fixes or enhances a compiled AML system. System management in AML incorporates patches for automatic loading. Patching a system allows incremental minor enhancements to an AML system without the need to recompile a system. Once patch files have been created, the files will be loaded automatically when a system is loaded. If the system is already loaded and a new patch is created it can be loaded without loading the complete system by using the function *patch-system*. Patch-system will load patches of the system version that is currently loaded into memory.

```
;; patch-system [FUNCTION] ( system-name )
> (patch-system :system)
;; Loading file "system\SYSTEM-1\patches\bins\patch-0001.btc"
```

C.7 aml-init.tsi

The `aml-init.tsi` file is located in the AML working (or startup) directory on WINDOWS platforms and in the user's home directory on UNIX platforms. The file is executed after AML interpreter is initialized.

```
(in-package :aml)

;; Loading patches
(patch-system :aml)

;; Loading modules
(load-module "aml-analysis-module-pack-type-3" :path "C:\\Program
Files\\Technosoft\\AML\\AML5.85_x64\\modules\\")
```

C.8 logical.pth

AML has the capability of defining logical-path-reference variables to locate resources on the file system. They are defined in logical paths files. On Unix platform, a user logical paths file (logical.paths) can be created under the user's home directory. On WINDOWS platforms, a logical paths file (`logical.pth`) exists under the AML working (or startup) directory and the user can append entries to it.

The logical paths file contains lines with logical-path-reference and corresponding path definitions.

```
:aml                     "C:\Program Files\Technosoft\AML\AML5.85_x64\"
:tmp                    c:\temp\
:temp                   c:\temp\
:home                   :aml
:lib                    :aml lib\
:patches                :aml patches\
:bitmaps                :aml bitmaps\
:ui-bitmaps             :aml gui\bitmaps\
:visual-object-editor            :aml visual-object-editor\
:ts-server-home         :aml pserver\
:geometry-exchange      :aml  geometry-exchange\
:geometry-exchange-exec :geometry-exchange  bin\
:model-save              :aml model-save\
:model-load-exec         :model-save bin\
:additional-modules      :aml modules\

:mechanism-system
"C:\Users\Rasmus\Documents\NTNU\Masteroppgave\code\mechanism-model\"

:nastran-path           "C:\Program Files\Siemens\NX 9.0\NXNASTRAN\bin"
```

```
:nastran-data
     "C:\Users\Rasmus\Documents\NTNU\Masteroppgave\nastran-data"
```

C.9 Source code management

The management of source code is accomplished through the definition of systems. A system is a set of source code files that are grouped together. Defining a system allows the code in a system to be treated as a module that may be loaded, compiled, and archived as a single entity. Compiling a system archives the source code with the binary files so updating older versions is possible by using the archived code. A system also compiles binaries for multiple platforms within a version to allow different platforms to be operating with the same system version. A system may require other systems to automatically load before loading or compiling itself. Organizing code into systems that may be loaded is a methodology for the reuse of code. A logical path is a reference to files and directories in the system. AML has the capability of defining logical-path-reference variables to locate resources on the file system. They are defined in logical paths files. The logical path file stores the logical path references, making modification easy. These references are converted by the use of the *logical*-path function. The `logical.pth` file is typically located in the working directory of the AML process. The *logical-path* function uses the logical paths file to convert a logical-path-reference to a path definition. The path definition is a string that is retrieved from the logical paths file.

The define-system construct is the main mechanism for creating systems. The definition should appear in a system definition file named system.def.

```
(define-system :MY-SYSTEM
     :require-systems '(:base-system :extension-system)
     :files '(
          "file1.aml"
          "file2.aml"
          "file3.aml"
          )
     )
```

The system.def must be in the directory returned when *(logical-path :my-system)* is executed. The *:my-system* must be an entry in the logical path file for the system to be found for compiling or loading.

```
:my-system /home/apps/my-system/
```

A directory structure for a system used in this thesis, named my-system, looks like the following:

```
      /home/
            apps/
                  my-system/
                        system.def ;; File containing the system definition.
                  sources/
                        file1.aml ;; Source code file.
                        file2.aml ;; Source code file.
                        file3.aml ;; Source code file.
```

When a system is compiled the source code is read from the files specified in define-system. Compilation of a system will create system versions (archives) that contain the source from time of compile and the binary files created by those source files. Binary files are created in a subdirectory named for the machine type in the system version subdirectory. That subdirectory will be used by load-system to load the binary files. A system tracks the binary files created and will not compile source files that have not changed since the last being compiled unless the force? keyword is t. Only the newest version or a new version may be compiled. The following compiles the system files the first time on a Windows; machine and creates MY-SYSTEM-1 subdirectory structure shown below

```
      > (compile-system :my-system)
```

New directory structure, creating a new subdirectory of my-system.

```
/home/
      apps/
            my-system/
                  system.def File containing the system definition.
            sources/
                  file1.aml Source code file.
                  file2.aml Source code file.
                  file3.aml Source code file.
            MY-SYSTEM-1/
                  system.def
                  sources/
                        file1.aml Archived source code file.
                        file2.aml Archived source code file.
                        file3.aml Archived source code file.
                  bins64/
                        compilation.dat ;; Archived binary patch files
                        file1.btc64 Archived binary file.
                        file2.btc64 Archived binary file.
                        file3.btc64 Archived binary file.
```

When a system is loaded the binary files that were created during the last compilation are loaded if no version number is supplied. When a version number is supplied the binaries for the machine will be loaded from a compile that may not be the newest. This allows versions to be in production and newer versions to be under development. If the source code is changed the changes will not be loaded until after a *compile-system* is performed. A system also tracks the version of the binary files that are loaded so that successive loading of the same system will not take time load files that are unchanged.

Running previously written AML-programs cannot be done without first starting the AML interpreter. A system can be loaded and compiled on startup by editing the `logical.pth` file or with commands directly in the AML interpreter. But running previously written AML-programs with batch scripts is not possible since the batch-process cannot interact with the interpreter after it is launched.

Appendix D　Source code

```
;;;----------------------------------------------------
;;; System : :mechanism-system
;;; Purpose : AML Mechanism Model
;;;
;;;
;;; Author : Rasmus Korvald Skaare
;;;

(in-package :AML)

(defvar #MECHANISM-LIBRARY# "")
(setf #MECHANISM-LIBRARY# (logical-path :mechanism-system "library"))

(define-system :mechanism-system
  :files '(
           "data-models.aml"
           "constraints.aml"
           "cross-sections.aml"
           "analysis.aml"
           "links.aml"
           "collections.aml"
           )
  )
```

```
;; Use point-object to define position properties
(define-class point-data-model
  :inherit-from (point-object data-model-node-mixin)
  :properties (
               (coordinates :class 'editable-data-property-class
;;;                    label "Coordinates"
;;;                    formula '(0 0 0)
                   )

               property-objects-list (list
                                       (list (the superior coordinates self)
                                             '(apply-formula? t))
                                       )
               )
  )

;; Direction vector
(define-class vector-data-model
  :inherit-from (vector-class data-model-node-mixin)
  :properties (
               ;;superior reference traverse
               point-ref (default ^point-default)
               (point-default :class 'point-data-model
                   coordinates '(0 0 0)
                   )
               (direction :class 'editable-data-property-class
;;;                    label "Direction"
                   formula (default)
                   )
               length 0.2

               base-point (the coordinates (:from (the superior point-ref)) 
)
               ;;sec-point (add-vectors ^base-point (multiply-vector-by-scal
ar ^direction ^length))

               property-objects-list (list
                                       (list (the coordinates self (:from (th
e superior point-ref)) )
                                             '(apply-formula? t))
                                       (list (the superior direction self)
                                             '(apply-formula? t))
                                       )


               )
  )


;; Frame, coordinate system
(define-class frame-data-model
  :inherit-from (coordinate-system-class data-model-node-mixin)
  :properties (
               ;;traverse to superior reference
               point-ref (default (the point-default (:from ^z-vector-ref)) 
)
               z-vector-ref (default ^z-vector-default)
               (z-vector-default :class 'vector-data-model
                   direction '(0 0 1)
                   )
               x-vector-ref (default ^x-vector-default)
               (x-vector-default :class 'vector-data-model
                        direction '(1 0 0)
                        )
                vector-k (the direction (:from ^z-vector-ref))
                vector-i (the direction (:from ^x-vector-ref))
                origin (the coordinates (:from ^point-ref))
                vector-j (cross-product ^vector-k ^vector-i)
                length 0.1

                property-objects-list (list
                                        (list  (the coordinates self (:from ^p
oint-ref) )
                                              '(apply-formula? t))
                                        (list (the direction self (:from ^z-ve
ctor-ref) )
                                              '(apply-formula? t))
                                        (list  (the direction self (:from ^x-v
ector-ref) )
                                              '(apply-formula? t))
                                        )
                )
  )

;;; (define-class su-vector-model
;;;   :inherit-from (vector-data-model)
;;;   :properties(


;;; (define-class su-frame-model
;;;   :inherit-from (frame-data-model)
;;;   :properties(
;;;                 vector-k (the direction (:from (the superior
;;; ------------------------------
;;; CONSTRAINT DEFINITIONS
;;; ------------------------------

(define-class general-joint-solid-model
  :inherit-from (frame-data-model)
  :properties (
               max-element-size 0.004
               scale-factor (default 1.2)

               point-ref (default nil)
               direction (default nil)
               (z-vector-ref :class 'vector-data-model
                   direction ^^direction
                   )
               (x-vector-ref :class 'vector-data-model
                   direction (arbitrary-normal-to-vector ^^direction)
                   )

               constraint-type (default nil)

               from-link (nth (position (nth 0 ^link-incidence) ^^link-list)
 ^^link-ref-list)
               to-link (nth (position (nth 1 ^link-incidence) ^^link-list) ^
^link-ref-list)

               link-incidence (default nil)

               link-ref-incident-on-constraint nil

               constraint-variable (default (list ))
               )
  )

(define-class master-joint-model
  :inherit-from (general-joint-solid-model)
  :properties (

               max-width (* ^scale-factor (max (first (the local-max-dimensi
on (:from ^female-element-solid-model))) (first (the local-max-dimension (:f
rom ^male-element-solid-model)) ) ) )
               max-height (* ^scale-factor (max (second (the local-max-dimen
sion (:from ^female-element-solid-model))) (second (the local-max-dimension 
(:from ^male-element-solid-model)) ) ) )

               link-ref-incident-on-constraint (list
                                                (the link-ref-incident-on-co
nstraint (:from ^male-element-solid-model))
                                                (the link-ref-incident-on-co
nstraint (:from ^female-element-solid-model))
                                                )
               )
  :subobjects (
               (male-element-solid-model :class (read-from-string (concatena
te !constraint-type "-male-element"))

                   )
               (female-element-solid-model :class (read-from-string (concate
nate !constraint-type "-female-element"))
                   (point-ref :class 'point-data-model
                              coordinates (convert-coords ^^superior '(0 0 0
) :from :local :to :global)
```

```
                              )
                   direction (convert-vector ^superior '(0 0 1) :from :local↵
 :to :global)

                   )
               )
  )
(define-class sub-point-model
  :inherit-from (point-data-model)
  :properties (
               reference-object ^main-frame
               coordinates '(0 0 0)
               )
  )
(define-class sub-frame-model
  :inherit-from (frame-data-model)
  :properties (
               (point-ref :class 'point-data-model
                   coordinates (convert-coords ^sub-point-ref (the coordinat↵
es (:from ^sub-point-ref)))
;;;                    coordinates (add-vectors (the coordinates (:from ^poi↵
nt-ref))
;;;                                                  (multiply-vector-by-scalar (↵
normalize ^direction) 0))
                   )
               )
  )

(define-class joint-element-model
  :inherit-from (general-joint-solid-model)
  :properties (
               union-list nil
               difference-list nil

               display? t
               gender (read-from-string (remove "-element-solid-model" (writ↵
e-to-string (object-name !superior))))
               gender_int (case ^gender ('male 0) ('female 1) )
               link-incidence (nth ^gender_int ^^link-incidence)
               link-ref-incident-on-constraint (get-link-ref ^links ^link-in↵
cidence)

               local-joint-index (position !superior (the constraints-incide↵
nt-on-link-list (:from ^link-ref-incident-on-constraint)) )
               members-connected-to-joint-element (loop for member in (the m↵
embers-ref-list (:from (the link-solid-geometry (:from ^link-ref-incident-on↵
-constraint)) ))
                                                     when (position !superior↵
 (the connection-between-constraints (:from member)))
                                                     collect member
                                                     )
               local-max-dimension (loop for member in (the members-ref-list↵
 (:from (the link-solid-geometry (:from ^link-ref-incident-on-constraint))))
                                     for pos = (position !superior (the conn↵
ection-between-constraints (:from member)))
                                     when (and pos (the display? (:from memb↵
er)) )
                                     maximize (if (= pos 0)
                                                  (max-width (the cross-sect↵
ion_D (:from member)) )
                                                (max-width (the cross-sectio↵
n_A (:from member)) ) ) into max-w
```

```
                                      and  maximize (if (= pos 0)
                                                     (max-height (the cros
s-section_D (:from member)) )
                                                   (max-height (the cross-
section_A (:from member)) ) ) into max-h
                                      finally (return (list max-w max-h))
                                      )

                  (sub-point-ref :class 'sub-point-model
                      )
                  )
   :subobjects (
                  (main-frame :class 'frame-data-model
                      ;; inherited frame properties used in main-frame and sub-
frame
                     (x-vector-ref :class 'vector-data-model
                                  direction (let (
                                        (first-sweep (nth 0 ^members-connected
-to-joint-element))
                                        (x-dir  (subtract-vectors
                                                 (the coordinates (:from (the 
point-ref (:from (nth 0 (the connection-between-constraints (:from first-swe
ep) ) )))))
                                                 (the coordinates (:from (the 
point-ref (:from (nth 1 (the connection-between-constraints (:from first-swe
ep) ) )))))
                                                 ))
                                        (x-dir-normal (cross-product ^^directi
on (cross-product ^^direction x-dir)))
                                        )
                                        (if (equal 0 (vector-length x-dir-norm
al))
                                            x-dir-normal
                                          (arbitrary-normal-to-vector ^^direct
ion)
                                          )
                                   )
                                 )
                      )

                  (sub-frame :class 'sub-frame-model
                      )
                  (joint-center-node :class 'mesh-node-class
                      coordinates (the origin (:from ^main-frame))
                      mesh-object (the link-mesh (:from (the link-mesh-model (:
from ^link-ref-incident-on-constraint))))
                      )
                  )
  )

(define-method get-constraint-incidence master-joint-model (link-index)
  (let (
        (pos (position link-index !link-incidence) )
        )
    (when pos
      (nth pos (children (the) :class 'general-joint-solid-model))
      )
    )
  )

(define-method get-constraint-incidence joint-element-model (link-index)
 (when (member link-index ^link-incidence) (the self))
 )


(define-class open-constraint-class
  :inherit-from (master-joint-model)
  :properties (
               )
  )
(define-class open-male-element
  :inherit-from (joint-element-model)
  :properties (
               (sub-point-ref :class 'sub-point-model
                   orientation (list
                                 (translate (list 0 0 (/ (- ^^max-height) 1) 
))
                                 )
                   )
               )
  )
(define-class open-female-element
  :inherit-from (joint-element-model)
  :properties (
               (sub-point-ref :class 'sub-point-model
;;;                      orientation (list
;;;                                    (translate (list 0 0 (/ ^^max-height 2) 
))
;;;                                    )
                   )
               )
  )

(define-class rigid-constraint-class
  :inherit-from (master-joint-model)
  :properties (
               )
  )


(define-class helical-constraint-class
  :inherit-from (master-joint-model)
  :properties (
;;;                joint-variables 'theta or 's
               ;;;                  to element
;;;                (temp-element :class 'vector-data-model
;;;                    point-ref  ^^point-ref
;;;                    direction  ^^direction
                   orientation (list
                                  (translate !direction :distance (nth 0 !co
nstraint-variable))
                                  )
;;;                    )
               )

  )




(define-class prismatic-constraint-class
  :inherit-from (master-joint-model)
  :properties (
;;;                joint-variables 's
```

```
               ;;;                to element
;;;                (temp-element :class 'vector-data-model
;;;                    point-ref   ^^point-ref
;;;                    direction   ^^direction
                   orientation (list
                                  (translate !direction :distance (nth 0 !co
nstraint-variable))
                                  )
;;;                    )
               )
  )

(define-class cylindric-constraint-class
  :inherit-from (master-joint-model)
  :properties (
;;;                joint-variables 'theta and 's
               ;;;                to element
;;;                (temp-element :class 'vector-data-model
;;;                    point-ref   ^^point-ref
;;;                    direction   ^^direction
                   orientation (list
                                  (translate !direction :distance (nth 0 !co
nstraint-variable))
                                  )
;;;                    )
               )
  )

(define-class planar-constraint-class
  :inherit-from (master-joint-model)
  :properties (
;;;                joint-variables 's, s' and  theta

;;;                (temp-element :class 'vector-data-model
;;;                    orientation (list
;;;                                  (translate
;;;                                   (angle-between-2-vectors (the superior
solid-element direction) (nth 0 ^^constraint-variable) )
;;;                                   (cross-product (the superior solid-elem
ent direction)  (nth 0 ^^constraint-variable) )
;;;                                   :axis-point (the coordinates (:from (th
e superior solid-element point-ref)) )
;;;                                   )
;;;                                  )
;;;                    )
               )
  )

(define-class revolute-constraint-class
  :inherit-from (master-joint-model)
  :properties (
               diameter 0
               ;;; dimensions to make sure connected joints match
;;;                diameter 0
;;;                joint-variables 'theta
               )
  )


(define-class revolute-male-element
  :inherit-from (joint-element-model)
  :properties (
               union-list (when (and (plusp ^max-width) (plusp ^max-height) 
)
                             (list ^imprinted-pin ^eye)
                             )
;;;                union-list (list ^pin ^eye)
               object-list (list ^imprinted-pin ^eye)
               simplify? nil
;;;                reference-object (the superior superior)
               (imprinted-pin :class '(tagging-object geometry-with-split-pe
riodic-faces-class)
                   source-object ^^pin
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   )

               (pin :class 'cylinder-object
                   reference-object ^main-frame
                   height (* 2 ^^max-height ^^scale-factor)
                   diameter (/ ^^max-width 2)
                   orientation (list
                                 (translate (list 0 0 (- (/ ^height 4)) ) )
                                 )
                   )
               (eye :class '(tagging-object cylinder-object)
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   reference-object ^sub-frame
                   height ^^max-height
                   diameter ^^max-width
;;;                    orientation (list
;;;                                  (translate (list 0 0 0) )
;;;                                  )
                   )
               (sub-point-ref :class 'sub-point-model
                   orientation (list
                                 (translate (list 0 0 (/ (- ^^max-height) 1) 
))
                                 )
                   )
               )
  :subobjects (

               (mating-surface-nodes :class 'mesh-nodes-query-class
                   tagged-object-list (list ^^imprinted-pin)
                   mesh-object (the link-mesh (:from (the link-mesh-model (:
from ^link-ref-incident-on-constraint))))
                   color 'green
                   )
               )
  )
(define-class revolute-female-element
  :inherit-from (difference-object joint-element-model)
  :properties(
              union-list (when (and (plusp ^max-width) (plusp ^max-height) )
                           (list ^fork)
                           )
              difference-list (when (and (plusp ^max-width) (plusp ^max-heig
ht) )
                                 (list ^imprinted-pin-hole)
                                 )
              object-list (list ^fork ^imprinted-pin-hole)
;;;                reference-object (the superior superior)

               (imprinted-pin-hole :class '(tagging-object geometry-with-spl
it-periodic-faces-class)
                   source-object ^^pin-hole
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   )

               (pin-hole :class 'cylinder-object
                   reference-object ^main-frame
                   height (* 4 ^^max-height ^^scale-factor)
                   diameter (/ ^^max-width 2)
;;;                    orientation (list
;;;                                  (translate (list 0 0 (/ ^height 4)))
;;;                                  )
                   )

               (fork :class '(tagging-object cylinder-object)
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   reference-object ^sub-frame
                   height ^^max-height
                   diameter ^^max-width
;;;                    orientation (list
;;;                                  (translate (list 0 0 (* ^height ^^scale-
factor) ))
;;;                                  )
                   )

               (sub-point-ref :class 'sub-point-model
;;;                    orientation (list
;;;                                  (translate (list 0 0 (/ ^^max-height 2) 
))
;;;                                  )
                   )
               )
  :subobjects (

               (mating-surface-nodes :class 'mesh-nodes-query-class
                   tagged-object-list (list ^^imprinted-pin-hole)
                   mesh-object (the link-mesh (:from (the link-mesh-model (:
from ^link-ref-incident-on-constraint))))
                   color 'green
                   )
               )
  )


(define-class spheric-constraint-class
  :inherit-from (master-joint-model)
  :properties (

;;;                joint-variables 'theta, theta' and  theta''
               ;;;                to element
;;;                (temp-element :class 'vector-data-model
;;;                    point-ref  ^^point-ref
;;;                    direction  ^^direction
                   orientation (list
                                (rotate
                                 (angle-between-2-vectors ^direction (nth 0 
^constraint-variable) )
                                 (cross-product ^direction (nth 0 ^constrain
t-variable) )
                                 :axis-point (the coordinates (:from ^point-
ref) )
                                 )
                                )
;;;                     )
               )
  )

(define-class spheric-male-element
  :inherit-from (union-object joint-element-model)
  :properties (
               tag-dimensions '(1 2 3)
               tag-attributes (list ^^max-element-size .1
                                    0 0.1 0 20.0 1.0e-5)
               union-list (when (and (plusp ^max-width) (plusp ^max-height) 
)
                            (list ^imprinted-stud ^imprinted-ball ^plate)
                            )
               object-list (list ^imprinted-stud ^imprinted-ball ^plate)

               (imprinted-stud :class '(tagging-object geometry-with-split-p
eriodic-faces-class)
                   source-object ^^stud
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   )
               (stud :class 'cylinder-object
                   reference-object ^sub-frame
                   height (vector-length (subtract-vectors (the origin (:fro
m ^sub-frame)) (the origin (:from ^main-frame)) ) )
                   diameter (/ (the diameter (:from ^ball)) 2)
                   orientation (list
                                (translate (list 0 0 (/ ^height 2)) )
                                )
                   )
               (imprinted-ball :class '(tagging-object geometry-with-split-p
eriodic-faces-class)
                   source-object ^^ball
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   )
               (ball :class 'sphere-object
                   reference-object ^main-frame
                   diameter (* 3 (/ ^^max-width 4))
                   )
               (plate :class '(tagging-object cylinder-object)
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   reference-object ^^sub-frame
                   diameter ^^max-width
                   height ^^max-height
                   )

               (sub-point-ref :class 'sub-point-model
                   orientation (list
                                (translate (list 0 0  (- 0 (/ ^^max-height 2↵
) (* 1 (the diameter (:from ^^ball))) )) )
                                )
                   )
               )
  :subobjects (
;;;                (sub-frame :class 'frame-data-model
;;;                    reference-object ^^main-frame
;;;                    point-ref (the point-default (:from ^z-vector-ref))

               (mating-surface-nodes :class 'mesh-nodes-query-class
                   tagged-object-list (list ^^imprinted-ball)
                   mesh-object (the link-mesh (:from (the link-mesh-model (:↵
from ^link-ref-incident-on-constraint))))
                   color 'green
                   )
               )
  )

(define-class spheric-female-element
  :inherit-from (difference-object joint-element-model)
  :properties (
               union-list (when (and (plusp ^max-width) (plusp ^max-height) ↵
)
                             (list ^imprinted-socket)
                             )
               difference-list (when (and (plusp ^max-width) (plusp ^max-hei↵
ght) )
                                 (list ^imprinted-hole)
                                 )
               object-list (list ^imprinted-socket ^imprinted-hole)

               (imprinted-hole :class '(tagging-object geometry-with-split-p↵
eriodic-faces-class)
                   source-object ^^hole
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   )
               (hole :class 'sphere-object
                   reference-object ^sub-frame
                   diameter (* 3 (/ ^^max-width 4))
                   )
               (imprinted-socket :class '(tagging-object geometry-with-split↵
-periodic-faces-class)
                   source-object ^^socket
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   )

               (socket :class 'intersection-object
                   object-list (list ^sphere ^cyl)
                   reference-object ^sub-frame
                   )

               (sphere :class 'sphere-object
                   diameter ^^max-width
                   )

               (cyl :class 'cylinder-object
                   height ^^max-height
                   diameter ^^max-width
                   )
               )
  :subobjects (
               (mating-surface-nodes :class 'mesh-nodes-query-class
                   tagged-object-list (list ^^imprinted-hole)
                   mesh-object (the link-mesh (:from (the link-mesh-model (:
from ^link-ref-incident-on-constraint))))
                   color 'green
                   )
               )
  )


(define-class solid-knuckle90-male-constraint
  :inherit-from (union-object joint-element-model)
  :properties (
               union-list (when (and (plusp ^max-width) (plusp ^max-height) 
)
                            (list !superior)
                            )
;;;                union-list (list ^pin ^eye)
               object-list (list ^imprinted-pin ^eye)
               simplify? t
;;;                reference-object (the superior superior)
               (imprinted-pin :class '(tagging-object geometry-with-split-pe
riodic-faces-class)
                   source-object ^^pin
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   )

               (pin :class 'cylinder-object
                   reference-object ^main-frame
                   diameter (* 2 ^^max-height ^^scale-factor)
                   height (/ ^^max-width 2)
;;;                    orientation (list
;;;                                  (translate (list 0 0 (/ ^height 4)) )
;;;                                  )
                   )
               (eye :class '(tagging-object cylinder-object)
                   tag-dimensions '(1 2 3)
                   tag-attributes (list ^^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   reference-object ^sub-frame
                   diameter ^^max-height
                   height ^^max-width
;;;                    orientation (list
;;;                                  (translate (list 0 0 0) )
;;;                                  )
                   )

                )
  :subobjects (
               (sub-frame :class 'frame-data-model
                   reference-object ^^main-frame
                   point-ref (the point-default (:from ^z-vector-ref))
                   orientation (list
                                 (translate (list 0 0 (/ (- ^^max-height) 2) 
))
```

```
                                )
                   )
               (mating-surface-nodes :class 'mesh-nodes-query-class
                   tagged-object-list (list ^^imprinted-pin)
                   mesh-object (the link-mesh (:from (the link-mesh-model (:⏎
from ^link-ref-incident-on-constraint))))
                   color 'green
                   )
               )
  )
;;; -----------------------------
;;; END CONSTRAINT DEFINITIONS
;;; -----------------------------
```

```
;;Superclass used for querying class-names
(define-class section-model
  :inherit-from(tagging-object position-object)
  :properties (
               ;; must be set on init from parent object

               max-element-size 0.004
               tag-dimensions '(1 2)
               tag-attributes (list ^max-element-size .1
                                    0 0.1 0 20.0 1.0e-5)
;;;                width (set-initial-dimension (the self) :default-value 0.
04 )
;;;                height (set-initial-dimension (the self) :default-value 0
.04 )

               )
  )
(define-method max-width section-model ()
  !width
  )

(define-method max-height section-model ()
  !height
  )

(define-method set-initial-dimension property-object (&key (property-list (l
ist (object-name (the self)))) (default-value 0.02) )
  (let ()
    (loop for property in property-list
      do
;;;       (change-value (the-list (list property 'self) :from ^^z-vector_A-r
ef) (max default-value (the-list (list property) :from ^^z-vector_A-ref)) )
;;;       (change-value (the-list (list property 'self) :from ^^z-vector_D-r
ef) (max default-value (the-list (list property) :from ^^z-vector_D-ref)) )
      )
    default-value
    )
  )

(define-class circular-section
  :inherit-from (imprint-class section-model)
  :properties (
               target-object ^disc
               tool-object-list (list ^p1)
;;;                diameter is average or max of width/height?
               diameter (average ^width ^height)
               (disc :class 'disc-object
                   diameter ^^diameter
                   )
               (p1 :class 'point-object
                   coordinates (list (/ ^diameter 2) 0 0)
                   )
               )
  )
(define-method max-width circular-section ()
  !diameter
  )
(define-method max-height circular-section ()
  !diameter
  )

(define-class circular-tube-section
  :inherit-from (imprint-class section-model)
  :properties (
               target-object ^diff-object
               tool-object-list (list ^p1 ^p2)

               outer-diameter (average ^width ^height)
;;;                20% of diameter
               thickness (* 0.2 ^outer-diameter)

               (p1 :class 'point-object
                   coordinates (list (/ ^outer-diameter 2) 0 0)
                   )
               (p2 :class 'point-object
                   coordinates (list (/ (- ^outer-diameter ^thickness) 2) 0 
0)
                   )

               (diff-object :class 'difference-object
                   object-list (list ^outer-circular ^inner-circular)
                   simplify? t
                   )

               (outer-circular :class 'circular-section
                   diameter ^outer-diameter
                   )
               (inner-circular :class 'circular-section
                   diameter (- ^outer-diameter ^thickness)
                   )
               )
  )
(define-method max-width circular-tube-section ()
  !outer-diameter
  )
(define-method max-height circular-tube-section ()
  !outer-diameter
  )

(define-class rectangle-section
  :inherit-from (imprint-class section-model )
  :properties (
;;;                inherit from section-model
               width (default 0.04)
               height (default 0.01)

               target-object ^sheet
               tool-object-list (list ^p1)
               (p1 :class 'point-object
                   coordinates (list (/ ^width 2) 0 0)
                   )
               (sheet :class 'sheet-object
                   width ^^width
                   height ^^height
                   )
               )
  )

(define-class rectangle-tube-section
  :inherit-from (section-model difference-object)
  :properties (
;;;                inherit from section-model
               width (default 0.04)
               height (default 0.01)
```

```
                object-list (list ^outer-rectangle ^inner-rectangle)
;;;                 20% of average of width/height?
                thickness (* 0.2 (average ^width ^height) )
                simplify? t

                (outer-rectangle :class 'sheet-object
                    width ^^width
                    height ^^height
                    )
                (inner-rectangle :class 'sheet-object
                    width (- ^^width ^thickness)
                    height (- ^^height ^thickness)
                    )
                )
  )

;;; (define-class square-section
;;;   :inherit-from (rectangle-section section-model)
;;;   :properties (
;;;                 height ^width
;;;                 )
;;;   )
;;; (define-class square-tube-section
;;;   :inherit-from (rectangle-tube-section section-model)
;;;   :properties (
;;;                 height ^width
;;;                 )
;;;   )

(define-class line-section
  :inherit-from (line-object section-model)
  :properties (
;;;                 inherit from section-model
                height (default 0.04)
                point1 (list 0 (- (/ ^height 2)) 0)
                point2 (list 0 (/ ^height 2) 0)
                )
  )
;; noe joint displacement for shell model
(define-method max-width line-section ()
  !height
  )
;;
(define-method max-height line-section ()
  0
  )

(define-class I-beam-section
  :inherit-from (union-object section-model)
  :properties (
;;;                 inherit from section-model
                width (default 0.04)
                height (default 0.04)

                object-list (list ^top-flange ^web ^bottom-flange)
                ;;10% of width/height
                flange-thickness (* 0.1 ^height)
                web-thickness (* 0.1 ^width)
                simplify? t

                (top-flange :class 'sheet-object
                    width ^^width
```

```
                        height ^flange-thickness
                        orientation (list
                                     (translate (list 0 (half ^^height) 0))
                                     )
                        )
                    (web :class 'sheet-object
                        width ^web-thickness
                        height ^^height
                        )
                    (bottom-flange :class 'sheet-object
                        width ^^width
                        height ^flange-thickness
                        orientation (list
                                     (translate (list 0 (- (half ^^height)) 0))
                                     )
                        )

                    )
  )

(define-method max-height I-beam-section ()
  (+ !flange-thickness !height)
 )

(define-class H-beam-section
  :inherit-from (union-object section-model)
  :properties (
;;;                     inherit from section-model
                   width (default 0.04)
                   height (default 0.04)

                   object-list (list ^left-flange ^web ^right-flange)
                   ;; 10% of width/height
                   flange-thickness (* 0.1 ^width)
                   web-thickness (* 0.1 ^height)
                   simplify? t


                    (left-flange :class 'sheet-object
                        width ^flange-thickness
                        height ^^height
                        orientation (list
                                     (translate (list (half ^^width) 0 0))
                                     )
                        )
                    (web :class 'sheet-object
                        width ^^width
                        height ^web-thickness
                        )
                    (right-flange :class 'sheet-object
                        width ^flange-thickness
                        height ^^height
                        orientation (list
                                     (translate (list (- (half ^^width)) 0 0))
                                     )
                        )
                    )
  )
(define-method max-width H-beam-section ()
  (+ !width !flange-thickness)
 )
```

```
(define-class hexagon-section
  :inherit-from (imprint-class section-model)
  :properties (
               target-object ^poly
               tool-object-list (list ^p1)
               R (/ (average ^width ^height) 2)
               (poly :class 'polygon-object
                   vertices (list
                              (list ^R 0 0)
                              (list (/ ^R 2) (- (/ (* ^R (sqrt 3)) 2)) 0)
                              (list (- (/ ^R 2)) (- (/ (* ^R (sqrt 3)) 2)) 0)
                              (list (- ^R) 0 0)
                              (list (- (/ ^R 2)) (/ (* ^R (sqrt 3)) 2) 0)
                              (list  (/ ^R 2)  (/ (* ^R (sqrt 3)) 2) 0)
                              )
                   dimension 2
                   )
               (p1 :class 'point-object
                   coordinates (list 0 (/ (* ^R (sqrt 3)) 2) 0)
                   )

               )
  )

(define-method max-width hexagon-section ()
  (* 2 !R)
  )
(define-method max-height hexagon-section ()
  (* !R (sqrt 3))
  )


(define-class polygon-section
  :inherit-from (polygon-object section-model)
  :properties (
               vertices '( (1 1 0) (-1 1 0) (-1 -1 0) (1 -1 0))
               dimension 2
               )
  )
```

```
(define-class link-mesh-class
  :inherit-from (object)
  :properties (
               mesh-object ^link-mesh
               geometry-model-object (default nil)
               (node-set :class 'analysis-node-set-class
                   query-objects-list (list (the nodes-query (:from ^^link-m
esh)))
                   )
               )
  :subobjects (
               (link-mesh :class 'paver-mesh-class
                   object-to-mesh ^^geometry-model-object
                   mesh-database-object ^^mesh-database
;;;                for solid mesh use:  element-shape :tet
                   element-shape :quadtri
                   solid-mesh? nil
                   )

               (link-mesh-surface-elements :class 'mesh-elements-2d-query-cl
ass
                   tagged-object-list (append
                                       (the union-list (:from ^^geometry-mod
el-object))
                                       (the visible-members-ref-list (:from 
 ^^geometry-model-object))
                                       )

                   mesh-object ^^link-mesh
                   )

               (link-mesh-solid-elements :class 'mesh-elements-3d-query-clas
s
                   tagged-object-list (append
                                       (the union-list (:from ^^geometry-mod
el-object))
                                       (the visible-members-ref-list (:from 
 ^^geometry-model-object))
                                       )
                   mesh-object ^^link-mesh
                   )

               (mating-surface-nodes :class 'mesh-nodes-query-class
                   tagged-object-list (the mating-surface-list (:from ^^geom
etry-model-object))
                   mesh-object ^^link-mesh
                   )


               (fixed-nodes :class 'mesh-nodes-query-class
                   tagged-object-list (list
                                       (the fixed-edge (:from ^^geometry-mod
el-object))
                                       )
;;;                    mesh-object ^^link-mesh
                   )
               (loaded-nodes :class 'mesh-query-nodes-from-interface-class
                   interface-object (the point-ref (:from (the constraint-el
ement (:from (nth 0 ^^solid-constraints-ref-list) ))))
                   mesh-object ^^link-mesh
                   )
               )
  )
(define-class analysis-link-model-class
  :inherit-from (analysis-model-class)
  :properties (
               mesh-model-object nil
               mesh-database-object ^^mesh-database

               z-load (default -100.0)
;;;                analysis-type :linear-static
               load-case-objects-list (list ^load-case-1)
               material-catalog-object ^material-catalog
               materials-list (list 'steel)
               property-set-objects-list (list ^link-material-properties)

               element-set-2d-objects-list (list ^link-analysis-surface-elem
ents)
               element-set-3d-objects-list (list ^link-analysis-solid-elemen
ts)


               mesh-object (the link-mesh (:from ^mesh-model-object))
               node-set-objects-list (list (the node-set (:from ^mesh-model-
object)))
               )
  :subobjects (
               (material-catalog :class 'material-catalog-class
                   )
               (link-material-properties :class 'analysis-property-set-2d-ty
pe-1-class
                   material-name "Steel"
                   thickness 0.3
                )

               (link-analysis-surface-elements :class 'analysis-element-set-
2d-type-1-class
                   query-objects-list (list
                                        (the link-mesh-surface-elements (:fro
m ^^mesh-model-object))
                                        )
                   property-set-object ^^link-material-properties
                )
               (link-analysis-solid-elements :class 'analysis-element-set-3d
-type-1-class
                   query-objects-list (list
                                        (the link-mesh-solid-elements (:from 
^^mesh-model-object))
                                        )
                   property-set-object ^^link-material-properties
                )
               (fixed-nodes-constraint :class 'analysis-constraint-displacem
ent-class
                   target-object (the fixed-nodes (:from ^^mesh-model-object
))
                   tx 0.0
                   ty 0.0
                   tz 0.0
                   mx 0.0
                   my 0.0
                   mz 0.0
                   )
               (nodal-load :class 'analysis-load-force-nodal-class
                        target-object (the loaded-nodes
                                        (:from ^^mesh-model-object))
                        load-vector (list 0.0 0.0 ^^z-load)
                        )
                   (load-case-1 :class 'analysis-load-case-class
;;;                          load-objects-list (list ^nodal-load)
;;;                          constraint-objects-list (list ^^fixed-nodes-constrain
t)
                    )
                   (nastran-interface :class 'nastran-analysis-class
                        analysis-model-object ^superior
                        model-name (the folder (:from ^^mechanism-selection))
                        nastran-file-name (concatenate (write-to-string  (object-
name ^^superior)) ".bdf")
                        nastran-version (nth 2 '(:nei-nastran :msc-nastran :nx-na
stran))
;;;                          analysis-directory (logical-path "C:\\Users\\Rasmus\\
Documents\\NTNU\\Masteroppgave\\forelesning_150326\\mesh\\simple-beam\\")
                        )
                   )
  )
```

```
;;replacing calculate-complete-graph-combi
(defun connection-combinations (n)
   (loop for j from 0 to (- n 2)
     append (loop for k from (1+ j) to (1- n)
                 collect (list j k)
            )
     )
   )
(defun list-combinations (p)
  (let (
        (l (if (typep p 'list) p (if (typep p 'fixnum) (loop for i from 0 to
 (1- p) collect i) (list ) ) ) )
        (n (length l))
        )
    (loop for j from 0 to (- n 2)
      append (loop for k from (1+ j) to (1- n)
                collect (list (nth j l) (nth k l))
                )
      )
    )
  )

(defun connections-on-constraints (n)
  (let (
        (combi (connection-combinations n))
        (tot-length (/ (* n (1- n) ) 2) )
        )
    (loop for c from 0 to (1- n)
      collect (loop for i from 0 to tot-length
                 when (member c (nth i combi))
                 collect i
                 )
      )
    )
  )

(defun sweep-loop-combinations (n)
  (let (
        (c-loops (3-edge-loop-combinations n) )
        (sweep-con (connection-combinations n) )
        )
    (loop for ci from 0 to (1- (length c-loops))
      for list-com = (list-combinations (nth ci c-loops))
      collect (loop for si from 0 to (1- (length list-com))
                 collect (position (nth si list-com) sweep-con)
                 )
      )
    )
  )

(defun 3-edge-loop-combinations (n)
   (loop for i from 0 to (- n 3)
     append (loop for j from (1+ i) to (- n 2)
                 append (loop for k from (1+ j) to (1- n)
                             collect (list i j k)
                             )
                 )
     )
   )
(defun list-3-subset-combinations (p)
  (let (
        (l (if (typep p 'list) p (if (typep p 'fixnum) (loop for i from 0 to
o (1- p) collect i) (list ) ) ) )
        (n (length l))
        )
    (loop for i from 0 to (- n 3)
      append (loop for j from (1+ i) to (- n 2)
               append (loop for k from (1+ j) to (1- n)
                        collect (list (nth i l) (nth j l) (nth k l))

                        )
               )
      )
    )
  )

(defun it-tolerance (d &optional (grade 5))
  (* (expt 10 (* 0.2 (- grade 1))) (+ (* 0.45 (expt d (/ 1 3))) (* 0.001 d))
)
  )

(define-class surface-data-model
  :inherit-from (tagging-object surface-thickened-class)
  :properties (
               tag-dimensions '(1 2)
               tag-attributes (list ^^max-element-size .1
                                    0 0.1 0 20.0 1.0e-5)
               (display? :class 'flag-property-class
                   formula  (loop for i in ^edge-combination
                              when (or (not (the display? (:from (nth i ^mem
bers-ref-list)))) (not (the connection geom (:from (nth i ^members-ref-list)
))) )
                              do (return nil)
                              finally (> ^thickness 0)
                              )
                   )
               edge-combination nil
               source-object ^surface
               ;;surface thickness 20% of smalles cross-section height
               thickness (* 0.2 (loop for i in ^edge-combination
                            minimize (max-height (the cross-section_D (:from
(nth i ^members-ref-list))))
                            ))
               front-thickness (/ ^thickness 2)
               back-thickness (/ ^thickness 2)
               render 'shaded
               color 'red

               property-objects-list (list
                                      (list (the superior display? self)
                                            '(automatic-apply? t)
                                            )
                                      )
               )
  :subobjects (
               (surface :class 'surface-from-three-edge-curves-class
                   edge-1-object (the connection (:from (nth (nth 0 ^edge-co
mbination) ^members-ref-list)) )
                   edge-2-object (the connection (:from (nth (nth 1 ^edge-co
mbination) ^members-ref-list)) )
                   edge-3-object (the connection (:from (nth (nth 2 ^edge-co
mbination) ^members-ref-list)) )
                   )
               )
  )
  )

(define-class surfaces-on-link-collection
  :inherit-from (series-object)
  :properties (
;;;                 closed-loops-combinations (3-edge-loop-combinations (leng
th ^sweeps-ref-list))
;;;                 closed-loops-combinations (3-edge-loop-combinations (leng
th ^constraints-incident-on-link-list))
                closed-loops-combinations (sweep-loop-combinations (length ^c
onstraints-incident-on-link-list))
                visible-members-index (loop for mem in ^visible-members-ref-l
ist
                                        collect (the index (:from mem))
                                        )
                valid-surface-loops (intersection ^closed-loops-combinations 
(list-3-subset-combinations ^visible-members-index))
                quantity (length ^valid-surface-loops)
                class-expression 'surface-data-model
                series-prefix 'surface
                init-form '(
                            edge-combination (nth ^index ^valid-surface-loops
)
                            )
                )
  )



;;; -----------------------------
;;; Sheth-Uicker DEFINITIONS
;;; -----------------------------

(define-class connection-model
  :inherit-from (nurb-curve-object)
  :properties (
                ;; traverse to superior reference for
;;;                 frame_D (the sub-frame (:from
;;;                 frame_A
                pij (convert-coords ^frame_D '(0 0 0) :from :local :to :globa
l)
                wij (convert-vector ^frame_D '(0 0 1) :from :local :to :globa
l)

                pjk (convert-coords ^frame_A '(0 0 0) :from :local :to :globa
l)
                wjk (convert-vector ^frame_A '(0 0 1) :from :local :to :globa
l)

                weight-points (append (the weight-list (:from ^shape-ref))
                                      (make-sequence 'list (- (length (the po
int-list (:from ^shape-ref))) (length (the weight-list (:from ^shape-ref))))
                                                     :initial-element 1)
                                      )
                start-point (the origin (:from ^frame_D))
                end-point (the origin (:from ^frame_A))

                middle-points (case ^line-config
                                ('paralell (let (
                                                 (start-tangent (add-vectors 
^start-point (multiply-vector-by-scalar (normalize ^perpendicular-dir) (half
 ^param_b) ) ))
```

```
                                                (end-tangent (add-vectors ^e
nd-point (multiply-vector-by-scalar (normalize ^perpendicular-dir) (- (half 
^param_b)) ) ))
                                                (start-weight (list 0.5))
                                                (end-weight (list 0.5))
                                                )
                                            (if (roughly-same-point start-ta
ngent end-tangent)
                                                (list (append start-tangent 
start-weight) )
                                              (list (append start-tangent st
art-weight) (append end-tangent end-weight) )
                                              )
                                            ) )
                               ;; if param_a == param_c && paramb=0
                               ('intersecting (let (
                                                    (center (nth 0 ^inter_po
ints ))

                                                    (middle-point (add-vecto
rs center (multiply-vector-by-scalar (normalize (add-vectors (subtract-vecto
rs ^start-point center) (subtract-vectors ^end-point center ))) ^param_a) ) 
)
                                                    (angle-start-middle (/ (
angle-between-2-vectors (subtract-vectors ^start-point center) (subtract-vec
tors middle-point center )) 2 ))
                                                    (start-tangent (add-vect
ors center (multiply-vector-by-scalar (normalize (add-vectors (subtract-vect
ors ^start-point center) (subtract-vectors middle-point center ))) (/ ^param
_a (cosd angle-start-middle))) ) )

                                                    (angle-middle-end (/ (an
gle-between-2-vectors (subtract-vectors middle-point center) (subtract-vecto
rs ^end-point center )) 2))
                                                    (end-tangent (add-vector
s center (multiply-vector-by-scalar (normalize (add-vectors (subtract-vector
s middle-point center) (subtract-vectors ^end-point center ))) (/ ^param_a (
cosd angle-middle-end))) ) )

                                                    (start-weight (list (sin
d (/ (angle-between-2-vectors
                                                                            
(subtract-vectors start-tangent ^start-point)
                                                                            
(subtract-vectors start-tangent middle-point)
                                                                            
) 2) ) ) )
                                                    (middle-weight (list 1))
                                                    (end-weight (list (sind 
(/ (angle-between-2-vectors
                                                                            
(subtract-vectors end-tangent middle-point)
                                                                            
(subtract-vectors end-tangent ^end-point)
                                                                            
) 2) ) ) )
                                                    )
                                                 (list (append start-tangent
 start-weight) (append middle-point middle-weight) (append end-tangent end-w
eight) )
                                                )
                                              )
```

```
                                )
               start-weight (list (append ^start-point (list 1)))
               end-weight (list (append ^end-point (list 1)))

               points (let (
                            (shape-points (loop for p-index in (the point-li
st (:from ^shape-ref))
                                                for w in ^weight-points
                                                collect (append (the coordinates
 (:from (nth p-index ^point-ref-list)) ) (list w)) ))
                            )
                        (if shape-points
                            (append ^start-weight shape-points ^end-weight)
                          (append ^start-weight ^middle-points ^end-weight)
                          )
                        )
               rational? t
               homogeneous? t
               degree 2

               line-config (line-pose (the superior))
               inter_points (inter_section (the superior) ^line-config )
               perpendicular-dir (perp-dir (the superior) ^line-config )

               param_a (vector-length (subtract-vectors (the origin (:from ^
frame_A)) (the origin (:from ^frame_B))))
               param_b (vector-length (subtract-vectors (the origin (:from ^
frame_B)) (the origin (:from ^frame_C))))
               param_c (vector-length (subtract-vectors (the origin (:from ^
frame_C)) (the origin (:from ^frame_D))))

               )
  :subobjects (
;;;                cross section at start of spline
               (spline-frame_start :class 'frame-data-model
;;;                     reference-object ^frame_D
                   point-ref ^point-ref_D
                   z-vector-ref ^z-vector-ref_D

                   (x-vector-ref :class 'vector-data-model
;;;                                  direction ^^perpendicular-dir
                                 direction (subtract-vectors (nth 1 ^points)
  (nth 0 ^points))
                                 )
                   )
;;;                cross section at end of spline
               (spline-frame_end :class 'frame-data-model
;;;                     reference-object ^frame_A
                   point-ref ^point-ref_A
                   z-vector-ref ^z-vector-ref_A

                   (x-vector-ref :class 'vector-data-model
;;;                                   direction ^^perpendicular-dir
                                 direction (subtract-vectors (nth (1- (lengt
h ^points)) ^points) (nth (- (length ^points) 2) ^points))
                                 )
                   )
               ;;augumented frames from SU-convention
               (frame_B :class 'frame-data-model
                   (point-ref :class 'point-data-model
                              coordinates (nth 0 ^^^inter_points)
                              )
                   (z-vector-ref :class 'vector-data-model
;;;                                  direction (the direction (:from ^^z-vec
tor-ref_D) )
                                 direction ^^wij
                                 )
                   (x-vector-ref :class 'vector-data-model
                                 direction ^^perpendicular-dir
                                 )
                   )

               (frame_C :class 'frame-data-model
                   (point-ref :class 'point-data-model
                               coordinates (nth 1 ^^^inter_points)
                               )
                   (z-vector-ref :class 'vector-data-model
;;;                                   direction (the direction (:from ^^z-ve
ctor-ref_A) )
                                 direction ^^wjk
                                 )
                   (x-vector-ref :class 'vector-data-model
                                  direction ^^perpendicular-dir
                                  )
                   )

               )
  )


;; Middle point
(defun m-point (p1 d1 p2 d2)
  (add-vectors p1 (proj_v d1 (multiply-vector-by-scalar (subtract-vectors p2
 p1) 0.5) ) )
  )
;; Closest point
(defun cl-point (p1 d1 p2 d2)
  (let (
        (n1x (cross-product d1 d2))
        (n1d (dot-product n1x (cross-product p2 d2)) )
        (n2d (dot-product n1x (cross-product p1 d2)) )
        (d1s (dot-product n1x n1x))

        (l1s (multiply-vector-by-scalar d1 (/ n1d d1s)))
        (l2s (multiply-vector-by-scalar d1 (/ n2d d1s)))

        )
    (add-vectors p1 (subtract-vectors l1s l2s))
      )
  )


;;determine configuration of two lines in relation too eachother
(define-method line-pose connection-model ()
  (let (
        (v0_1 (cross-product !pij !wij) )
        (v0_2 (cross-product !pij !wij) )
        (coplan  (* 0.5  (+ (dot-product !wij v0_2) (dot-product v0_1 !wjk))
 ))

        (normal-mag (vector-length (cross-product !wij !wjk) ) )
        (coincident (vector-length (cross-product (subtract-vectors !pjk !pi
j) !wij) ) )
        )
    (if (/= 0 coplan) 'skew (if (/= 0 normal-mag) 'intersecting (if (= 0 coi⏎
ncident) 'coincident 'paralell) ) )
    )
  )

;; Generalized closest points
;; if lines Gij Gjk are intersecting or skew: closest point
;; if lines Gij Gjk are coincident or parallel: mid-point
;; Calculate intersection between lines
(define-method inter_section connection-model (line-config)
   (case line-config
     ('skew
      (list (cl-point !pij !wij !pjk !wjk) (cl-point !pjk !wjk !pij !wij) )
      )
     ('intersecting
      (list (cl-point !pij !wij !pjk !wjk) (cl-point !pjk !wjk !pij !wij) )
      )
     ('coincident
      (list (m-point !pij !wij !pjk !wjk) (m-point !pjk !wjk !pij !wij) )
      )
     ('paralell
      (list (m-point !pij !wij !pjk !wjk) (m-point !pjk !wjk !pij !wij) )
      )
     )
   )
;;get Generalized perpendicular direction
(define-method perp-dir connection-model (line-config)
  (let (
        (cross (cross-product !wij !wjk) )
        (ortho-comp (orthogonal-projection-complement !wij (subtract-vectors⏎
 !pjk !pij)) )
        )
    (case line-config
      ('skew cross )
      ('intersecting cross )
      ('coincident (read-from-string (pop-up-text-prompt
                                      :nb-entries 1
                                      :title "Please specify direction"
                                      :prompt "Type in x-vector"
                                      :init-text "(1 0 0)"
                                      :x-offset (/ (nth 0 (get-screen-size))⏎
 2)
                                      :y-offset (/ (nth 1 (get-screen-size))⏎
 2))))
      ('paralell ortho-comp )
      )
    )
  )


;; the orthogonal projection of a vector b onto some vector a
;; pi_a(b)
(defun proj_v (a b)
  (multiply-vector-by-scalar a (/ (dot-product b a ) (dot-product a a )) )
  )

;; the orthogonal projection of a vector b into the orthogonal complement of⏎
 a of some vector a
;; tau_a(b)
(defun orthogonal-projection-complement (a b)
  (subtract-vectors b (proj_v a b) )
  )
```

```
;;; ------------------------------
;;; END Sheth-Uicker DEFINITIONS
;;; ------------------------------

(defun ellipse-find-minor-axis (a r theta)
  (/ (* r a (sin theta)) (sqrt (- (* a a) (* r r (cos theta) (cos theta)))))
  )

(define-class member-solid-model
  :inherit-from (tagging-object general-sweep-class)
  :properties(
              tag-dimensions '(1 2 3)
              tag-attributes (list ^^max-element-size .1
                                   0 0.1 0 20.0 1.0e-5)

              (display? :class 'flag-property-class
                  formula (when (the cross-section-type (:from ^shape-ref))
t)
                  )

              connection-between-constraints nil
              frame_D (the sub-frame (:from (nth 0 ^connection-between-const
raints) ))
              frame_A (the sub-frame (:from (nth 1 ^connection-between-const
raints) ))
              point-ref_D (the point-ref (:from ^frame_D ))
              point-ref_A (the point-ref (:from ^frame_A ))

              z-vector-ref_D (the z-vector-ref (:from ^frame_D ))
              z-vector-ref_A (the z-vector-ref (:from ^frame_A ))


              ;;unused property, should differentiate between commen-sweeps
on start and end constraint
              common-sweeps (loop for constraint in ^connection-between-cons
traints
                                collect (remove !superior (the link-ref-on-con
straint (:from constraint)))
                                )


              ;;cross section dimension, width 0.04 / height 0.04
              shape-ref nil
              width (nth 0 (the solid-dimensions (:from ^shape-ref)))
              height (if (< 1 (length (the solid-dimensions (:from ^shape-re
f))))
                         (nth 1 (the solid-dimensions (:from ^shape-ref)))
                       (nth 0 (the solid-dimensions (:from ^shape-ref)))
                       )
              width-end (if (< 2 (length (the solid-dimensions (:from ^shape
-ref))))
                              (nth 2 (the solid-dimensions (:from ^shape-ref))
)
                            (nth 0 (the solid-dimensions (:from ^shape-ref)))
                            )
              height-end (if (< 3 (length (the solid-dimensions (:from ^shap
e-ref))))
                              (nth 3 (the solid-dimensions (:from ^shape-ref))
)
                            (nth 1 (the solid-dimensions (:from ^shape-ref)))
                            )
              ;;; sweep parameters
              profile-objects-list (list
                                      ^cross-section_D
                                      ^cross-section_A
                                      )
              path-points-coords-list (list
                                         (the origin (:from ^frame_D))
                                         (the origin (:from ^frame_A))
                                         )
              profile-match-points-coords-list (list
                                                  (vertex-of-object ^cross-sec
tion_D)
                                                  (vertex-of-object ^cross-sec
tion_A)
                                                  )

              path-object ^connection
              tangential-sweep? t

              ;;; if two cross-sections, only nil works, with one cross-sect
ion t gives best mesh
              simplify? nil
              render 'shaded

              ;;; cross-section selection
              (cross-section-type :class 'option-property-class
                  label "Cross-section Type"
                  mode 'menu
                  formula (if (the cross-section-type (:from ^shape-ref))
                               (nth (position (write-to-string (the cross-sec
tion-type (:from ^shape-ref))) !labels-list)  !options-list)
                             (nth (position (write-to-string (the cross-secti
on-type (:from ^default-shape))) !labels-list)  !options-list) )
                  options-list (reverse (class-direct-defined-subclasses 'se
ction-model))
                  labels-list (loop for option in !options-list
                                  collect (remove "-section" (write-to-string 
option))
                                  )
                  )
              property-objects-list (list
                                       (list (the superior cross-section-type 
self)
                                             '(automatic-apply? t)
                                             )
                                       (the superior width self)
                                       (the superior height self)
                                       (the superior width-end self)
                                       (the superior height-end self)
                                       (list (the superior display? self)
                                             '(automatic-apply? t)
                                             )
                                       '("Draw..." (button1-parameters :draw b
utton3-parameters :draw)
                                         ui-work-area-action-button-class)
                                       '("Undraw..." (button1-parameters :undr
aw button3-parameters :undraw)
                                         ui-work-area-action-button-class)    

                                       )
              )
  :subobjects (
               (connection :class 'connection-model
                   )

               (cross-section_D :class !cross-section-type
                   reference-object (the spline-frame_start (:from ^connecti
on))
                   orientation (list
                                (rotate 90 :x-axis)
                                (rotate 90 :z-axis)
                                )
                   )
               (cross-section_A :class !cross-section-type
                   width ^width-end
                   height ^height-end
                   reference-object (the spline-frame_end (:from ^connection
))
                   orientation (list
                                (rotate 90 :x-axis)
                                (rotate 90 :z-axis)
                                )
                   )
               )
  )

(define-method work-area-button1-action member-solid-model (params)
  (case params
    (:draw
     (draw self :draw-subobjects? nil)
     )
    (:undraw
     (undraw self :subobjects? nil)
     )
    )
  )
(define-method work-area-button3-action member-solid-model (params)
  (case params
    (:draw
     (draw self :draw-subobjects? t)
     )
    (:undraw
     (undraw self :subobjects? t)
     )
    )
  )

(define-class members-on-link-collection
  :inherit-from (series-object)
  :properties (
;;;                 traverse to superior reference:
;;;                 ^connection-between-2-constraints-combinations
;;;                 ^shapes-on-link
;;;                 ^constraint-connection-combination

               quantity (length ^connection-between-2-constraints-combinatio
ns)
               class-expression 'member-solid-model
               series-prefix 'member
               init-form '(
                            connection-between-constraints (nth ^index ^^conn
ection-between-2-constraints-combinations)

                            shape-ref (nth ^index ^^shapes-on-link)
                            )
                )
  )


(define-class link-geometry-class
;;;   :inherit-from (tagging-object difference-object)
  :inherit-from (tagging-object geometry-with-split-periodic-faces-class)
  :properties (
               ;;used for geometry-with-split-periodic-faces-clas
               source-object ^difference-element

               max-element-size 0.005
               tag-dimensions '(1 2)
               tag-attributes (list ^max-element-size .1
                                    0 0.1 0 20.0 1.0e-5)


               default-shape (let(
                                   (def (loop for shape in (children ^^^shape
s :class 'shape-data-model)
                                          when (and
                                                (equal 'default (the sweep-in
dex (:from shape)))
                                                (equal ^link-index (the link-
ref (:from shape)))
                                                )
                                          do (return shape)
                                          )
                                        )
                                   )
                                (if def def ^^default-shape)
                                )
               shapes-on-link (let (
                                     (shape-list (make-sequence 'list (length
 ^constraint-connection-combination) :initial-element ^default-shape) )
                                     )
                                 (loop for shape in (children ^^^shapes :clas
s 'shape-data-model)
                                   when (and (equal ^link-index (the link-ref
 (:from shape))) (not (equal 'default (the sweep-index (:from shape)))) )
                                   do (replace shape-list (list shape) :start
1 (the sweep-index (:from shape)) )
                                   finally (return shape-list)
                                   )
                                 )

               constraint-connection-combination (connection-combinations (l
ength ^constraints-incident-on-link-list))

;;;                combination-on-constraint (connections-on-constraints (le
ngth ^constraints-incident-on-link-list))

               surfaces-ref-list (children ^surfaces :class 'surface-data-mo
del :test '(and !geom !display?))
               ;; only include sweeps which are not "turned off"
               members-ref-list (children ^sweeps :class 'member-solid-model
)
               visible-members-ref-list (children ^sweeps :class 'member-sol
id-model :test '!display?)
```

```
;;;               solid-constraints-ref-list (children ^solid-constraints :
class 'solid-constraint-model)

               union-list (loop for l in ^constraints-incident-on-link-list
                                 append (the union-list (:from l))
                                 )
               difference-list (loop for l in ^constraints-incident-on-link-
list
                                      append (the difference-list (:from l))
                                      )

               object-list (append
                                 (list ^union-element)
;;;                                  (list ^imprint-union-element)
                                 ^difference-list
                                 )
               simplify? nil

               (imprint-union-element :class '(tagging-object geometry-with-
split-periodic-faces-class)
                   tag-dimensions '(1 2)
                   tag-attributes (list ^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   source-object ^^union-element
                   )

               (imprint-constraint-points :class '(tagging-object imprint-cl
ass)
                   target-object ^^imprint-union-element
                   tool-object-list (loop for c in ^constraints-incident-on-
link-list
                                          collect (the point-ref (:from c))
                                          )
                   )
               (difference-element :class '(tagging-object difference-object
)
                   tag-dimensions '(1 2)
                   tag-attributes (list ^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   object-list (append
                                    (list ^union-element)
;;;                                 (list ^imprint-union-element)
                                    ^difference-list
                                    )
                   simplify? t

                   )

               (union-element :class '(tagging-object union-object)
                   tag-dimensions '(1 2)
                   tag-attributes (list ^max-element-size .1
                                        0 0.1 0 20.0 1.0e-5)
                   object-list (append  ^surfaces-ref-list
;;;                                          (list ^sewn-links)
                                        ^visible-members-ref-list
                                        ^^union-list)
                   simplify? t
                   )
               )
  :subobjects (
               (surfaces :class 'surfaces-on-link-collection
                   )
               (sweeps :class 'members-on-link-collection
                   )
               )
  )

(define-class link-model-class
  :inherit-from (object)
  :properties (
               ;; properties set from parent init-form
               constraints-incident-on-link-list nil
               connection-between-2-constraints-combinations (list-combinati⮰
ons ^constraints-incident-on-link-list)
               link-index nil

               )
  :subobjects (
               (link-solid-geometry :class 'link-geometry-class
                   )
               (link-mesh-model :class 'link-mesh-class
                   geometry-model-object ^link-solid-geometry
                   mesh-database ^^mesh-database
                   )
               (analysis :class 'analysis-link-model-class
                   mesh-model-object ^^link-mesh-model
                   )
               )
  )
```

```
(define-class collection-class
  :inherit-from (object)
  :properties (
               collection-type nil
               )
  )


(define-method read-from-file collection-class ()
  (let (
        (file-name (write-to-string !collection-type ))
        (file-path (logical-path (the path (:from ^mechanism-selection)) (co
ncatenate file-name ".txt") ) )
        (function-name (read-from-string (concatenate "read-" file-name "-fr
om-file")) )
        )
    (if (and
         file-path
         (stringp file-path)
         (probe-file file-path)
         )
        (with-open-file (file file-path :direction :input)
          (apply function-name (list file))
          )
      (progn
        (message (format nil "\"~a\" is not a valid file path." file-path) :
append? t)
        nil
        )
      )
    )
  )

;;; Constraint collection
(define-class constraint-collection
  :inherit-from (series-object collection-class)
  :properties (
               ;;Traverse to Superior reference
;;;                point-ref-list nil
               collection-type 'constraints
               constraint-list (read-from-file !superior)
               quantity (length ^constraint-list)
               class-expression '(read-from-string (concatenate (nth 1 (nth 
!index !constraint-list)) "-constraint-class"))
               series-prefix 'c
               init-form '(
                           point-ref (nth (nth 0 (nth !index ^constraint-lis
t)) ^point-ref-list)
                           constraint-type (nth 1 (nth !index ^constraint-li
st))
                           link-incidence (nth 2 (nth !index ^constraint-lis
t))
                           direction (normalize (nth 3 (nth !index ^constrai
nt-list)))
                           constraint-variable (nth 4 (nth !index ^constrain
t-list))
                           )
               )
  )

(defun read-constraints-from-file (stream)
  (when stream
    (loop for line = (read-line stream nil :eof)
      until (equal line :eof)
      for ls = (string-to-delimited-token-list line :delimiter #\tab :string⮰
-token? nil)
      for c-data = (list
                     (nth 0 ls) (nth 1 ls) (read-from-string (nth 2 ls)) (rea⮰
d-from-string (nth 3 ls)) (read-from-string (nth 4 ls))
                     )
      collect c-data
      )
    )
  )




(define-class point-collection
  :inherit-from (series-object collection-class)
  :properties (
;;;                 points-list (FORMATTED-list-FROM-FILE (logical-path MECHA⮰
NISM-LIBRARY "coordinates.txt") :element-format '(x y z))
               collection-type 'coordinates
               points-list (read-from-file !superior)
               quantity (length ^points-list)
               class-expression 'point-data-model
               series-prefix 'p
               init-form '(
                            coordinates (nth ^index ^points-list)
                            )
               )
  )

(defun read-coordinates-from-file (stream)
;;;   (with-open-file (file (logical-path (the path (:from (the superior mec⮰
hanism-selection)) ) "coordinates.txt") :direction :input)
  (when stream
    (loop for line = (read-line stream nil :eof)
      until (equal line :eof)
      for xyz = (read-from-string (format nil "(~a)" line))
      collect xyz
      )
    )
  )

(define-class shape-data-model
  :inherit-from (object)
  :properties (
               link-ref (default 'default)
               sweep-index (default 'default)
               cross-section-type (read-from-string (remove "-section" (writ⮰
e-to-string (default 'circular-section))))
               solid-dimensions '(0.04 0.04)
               point-list nil
               weight-list nil
               )
  )

(define-class shape-collection
  :inherit-from (series-object collection-class)
  :properties (
;;;                 points-list (FORMATTED-list-FROM-FILE (logical-path MECHA⮰
NISM-LIBRARY "coordinates.txt") :element-format '(x y z))
               collection-type 'shapes
```

```
                shapes-list (read-from-file !superior)
                quantity (length ^shapes-list)
                class-expression 'shape-data-model
                series-prefix 'shape
                init-form '(
                            link-ref (nth 0 (nth ^index ^shapes-list))
                            sweep-index (nth 1 (nth ^index ^shapes-list))
                            cross-section-type (nth 2 (nth ^index ^shapes-lis
t))
                            solid-dimensions (nth 3 (nth ^index ^shapes-list)
)
                            point-list (nth 4 (nth ^index ^shapes-list))
                            weight-list (nth 5 (nth ^index ^shapes-list))
                            )
                )
  )

;;; link-index sweep-index cross-section-type   (dimensions)   (points list)
  (weight-list) degree   (knot-list)
(defun read-shapes-from-file (stream)
;;;   (with-open-file (file (logical-path (the path (:from ^mechanism-select
ion) ) "shapes.txt") :direction :input)
 (when stream
   (loop for line = (read-line stream  nil :eof)
     until (equal line :eof)
     for ls = (string-to-delimited-token-list line :delimiter #\tab :string-
token? nil)
     for shape-data = (list
                        (nth 0 ls) (nth 1 ls) (read-from-string (nth 2 ls)) (
read-from-string (nth 3 ls)) (read-from-string (nth 4 ls)) (read-from-string
 (nth 5 ls))
                        )
     collect shape-data
     )
   )
 )

(define-class link-collection
  :inherit-from (series-object)
  :properties (
                (cross-section-type :class 'option-property-class
                    label "Cross-section Type"
                    mode 'menu
                    formula (nth 0 !options-list)
                    options-list (reverse (class-direct-defined-subclasses 's
ection-model))
                    labels-list (loop for option in !options-list
                                  collect (remove "-section" (write-to-string
 option))
                                  )
                    )
                common-width 0.04
                common-height 0.04
                property-objects-list (list
                                        (the superior cross-section-type self)

                                        '("Set all cross-sections" (button1-pa
rameters :set-c button3-parameters :unset)
                                          ui-work-area-action-button-class)

                                        (the superior common-width self)
                                        (the superior common-height self)
```

```
                                       '("Set all dimensions" (button1-parame⮰
ters :set-d button3-parameters :unset)
                                         ui-work-area-action-button-class)
                                       )
               ;;Traverse to Superior reference
;;;                point-ref-list nil
;;;                constraints-ref-list nil
               link-list (sort (copy-seq (remove-duplicates (append-list (lo⮰
op for kid in (the constraints-ref-list)
                                                                 collect (th⮰
e link-incidence (:from kid))
                                                                 )
                                                               ))
                                         ) '<)

               (init-default-shape :class 'shape-data-model
                   )

               default-shape (let(
                                   (def (loop for shape in (children ^^shapes⮰
 :class 'shape-data-model)
                                          when (equal 'default (the link-ref ⮰
(:from shape)))
                                          do(return shape)
                                          ) )
                                   )
                                (if def def ^init-default-shape)
                                )

               quantity (length ^link-list)
;;;                class-expression 'link-data-model
               class-expression 'link-model-class
               series-prefix 'link
               init-form '(
                           link-index (nth ^index ^^link-list)
                           constraints-incident-on-link-list (loop for kid i⮰
n (the constraints-ref-list)
                                                               for con = (ge⮰
t-constraint-incidence kid ^link-index)
                                                               when con coll⮰
ect con
                                                               )
;;;                            constraints-incident-on-link-list (select-obj⮰
ect :from (the superior superior constraints) :class 'constraint-model :test⮰
 '(member (nth ^index ^^link-list) (the link-incidence)))
                           )
               )
  )

(define-method get-link-ref link-collection (link-index)
  (nth (position link-index !link-list) ^link-ref-list)
  )

(define-method work-area-button1-action link-collection (params)
  (case params
    (:set-c
     (loop for l in ^link-ref-list
       do (loop for s in (the members-ref-list (:from (the link-solid-geomet⮰
ry (:from l)) ))
            do (change-value (the cross-section-type self (:from s)) !cross-⮰
section-type)
            )
       )
     )
    (:set-d
     (loop for l in ^link-ref-list
       do (loop for s in (the members-ref-list (:from (the link-solid-geomet
ry (:from l)) ))
            do (change-value (the width self (:from s)) !common-width)
            (change-value (the height self (:from s)) !common-height)
            )
       )
     )
    )
  )

(define-class folder-info-model
  :inherit-from (object)
  :properties (
               path nil
               folder (subseq (remove #MECHANISM-LIBRARY# ^path) 1)
               label (replace (copy-seq ^folder) " " :start1 (position "-" ^
folder :test 'string-equal) )

               class-name (let (
                                (name (read-from-string (concatenate ^folder
 "-class" ) ))
                                )
                            (when (find-class name)
                              name
                              )
                            )
               )
  )
(define-class folder-collection
  :inherit-from (series-object)
  :properties (

               ;; Removing ../ and ./
               library-subfolder-list (rest (rest (directory #mechanism-libr
ary#)) )
               quantity (length ^library-subfolder-list)
               class-expression 'folder-info-model
               series-prefix 'folder
               init-form '(
                           path (nth ^index ^^library-subfolder-list)
                           )
               )
  )


(define-class mechanism-collection
  :inherit-from (data-model-node-mixin)
  :properties (
               point-ref-list  (children ^points :class 'point-data-model )
               constraints-ref-list (children ^constraints :class 'master-jo
int-model)
               link-ref-list (children ^links :class 'link-model-class)

               (mechanism-selection :class 'option-property-class
                   labels-list (loop for subfolder in (children ^folders :cl
ass 'folder-info-model)
                                 when (the path (:from subfolder))
                                 collect (the label (:from subfolder))
                                )
;;;                    labels-list (select-object :from ^folders :class 'fol
der-info-model :test '(the path) :eval '(the label) )
                   options-list (children ^folders :class 'folder-info-model
)

                   mode 'menu
                   formula (nth (position "four bar" !labels-list) !options-
list)
                   )

               property-objects-list (list
                                      (list (the superior mechanism-selectio
n self)
                                            '(automatic-apply? t))
                                     )
;;;                Storage

               ;;Property storing folder from library
               (folders :class 'folder-collection
                   )

               write-to-file (write-save-fedem-solver !superior)
               )
  :subobjects (


               (constraints :class 'constraint-collection
                   )
               (points :class 'point-collection
                   )
               (links :class 'link-collection
                   )

               (shapes :class 'shape-collection
                   )

               (mesh-database :class 'meshdb-class
                   )
               )
  )

(define-method write-save-fedem-solver  mechanism-collection ()
  (let (
        (idcount 10)
        (extidcount 0)
        (superelements nil)
        (triads nil)
        )
    (with-open-file (stream (logical-path (the path (:from !mechanism-select
ion)) (concatenate "fedem_solver" ".fsi") )
                            :direction :output
                            :if-exists :overwrite
                            )
      (progn
        (loop for link in !link-ref-list
          do
          (setf idcount (1+ idcount))
          (setf extidcount (1+ extidcount))
          (setf superelements (append superelements (list idcount)))

          (loop for triad in (the solid-constraints-ref-list (:from link))
            for frame =  (vector-to-list (get-position-matrix (the main-fram
e (:from triad)) ) )
            when (not (and (equal 0 (the index (:from link))) (equal "open" 
(superior (the constraint-element (:from triad)))) ))
            do
            (setf idcount (1+ idcount))
            (setf extidcount (1+ extidcount))
            (format stream "~a~%" "&TRIAD")
            (format stream "~t id = ~d~%" idcount)
            (format stream "~t extId = ~d~%" extidcount)
            (format stream "~t nDOFs = ~d~%" (if (equal 0 (the index (:from 
link))) 0 6) )
            (format stream "~t ur = ~{~{~1,9e ~}~%~6t ~}"  (list (append (su
bseq frame 0 3) (list (nth 12 frame)))
                                                              (append (su
bseq frame 4 7) (list (nth 13 frame)))
                                                              (append (su
bseq frame 8 11) (list (nth 14 frame)))
                                                              ) )
            (format stream "~%/~2%")

            and collect idcount into trs
            finally (setf triads (append triads (list trs)))
            )

          )
        (print triads)
        (setf extidcount 0)
        (loop for link in !link-ref-list
          for id = (the index (:from link))
;;;           from 0 to (1- (length !link-ref-list))
          for triader = (nth id triads)
          for frame =  (vector-to-list (get-position-matrix (the main-frame 
(:from (nth 0 (the solid-constraints-ref-list (:from link)))) ) ) )
          when (not (equal 0 id ) )
          do
          (setf idcount (1+ idcount))
          (format stream "~a~%" "&SUP_EL")
          (format stream "~t id = ~d~%" idcount)
          (format stream "~t extId = ~d~%" id)
          (format stream "~t numGenDOFs = 0~%")
          (format stream "~t numTriads = ~d~%" (length triader))
          (format stream "~t triadIds = ~{~d ~}~%" triader)
          (format stream "~t shadowPosAlg = 1~%")
          (format stream "~t massCorrFlag = -1~%")
          (format stream "~t stiffScale = 1.0~%")
          (format stream "~t massScale = 1.0~%")
          (format stream "~t alpha1 = 0.0, alpha2 = 0.0~%")
          (format stream "~t subPos = ~{~{~1,9e ~}~%~6t ~}"  (list (append (
subseq frame 0 3) (list (nth 12 frame)))
                                                               (append (
subseq frame 4 7) (list (nth 13 frame)))
                                                               (append (
subseq frame 8 11) (list (nth 14 frame)))
                                                                        
      ) )
          (format stream "~%/~2%")
          (print (the solid-constraints-ref-list (:from link)) )
          (loop for triad in (the solid-constraints-ref-list (:from link))
            for i = (the index (:from triad) )
;;;             for un_frame = (vector-to-list (matrix-multiply (get-positio
n-matrix (the main-frame (:from triad))) (matrix-inverse frame) ))
```

```
            do
            (print i)
            (print triader)
            (format stream "~a~%" "&TRIAD_UNDPOS")
            (format stream "~t supElId = ~d~%" idcount)
            (format stream "~t triadId = ~d~%" (nth i triader) )
;;;             (format stream "~t undPosInSupElSystem = ~{~{~1,9e ~}~%~6t ~
}"  (list (append (subseq un_frame 0 3) (list (nth 12 un_frame)))
;;;
          (append (subseq un_frame 4 7) (list (nth 13 un_frame)))
;;;
          (append (subseq un_frame 8 11) (list (nth 14 un_frame)))
;;;
          ) )
            (format stream "~%/~2%")
           )
          )
        )
      )
    )
  )
```

Appendix E　Risk analysis

NTNU HMS/KS	Risikomatrise	utarbeidet av	Nummer	Dato
		HMS-avd.	HMSRV2604	08.03.2010
		godkjent av		Erstatter
		Rektor		09.02.2010

Enhet: IPM **Dato: 25.02.15**

Linjeleder: Torgeir Welo

Deltakere ved kartleggingen (m/ funksjon)**: Rasmus Korvald Skaare** (student)**, Ole Ivar Sivertsen** (veileder)
(Ansv. veileder, student, evt. medveiledere, evt. andre m. kompetanse)

Kort beskrivelse av hovedaktivitet/hovedprosess: Masteroppgave Rasmus Korvald Skaare. Mechanism Parametrization, Modeling and FE-Meshing.

Er oppgaven rent teoretisk? (JA/NEI)**: JA** *«JA» betyr at veileder innestår for at oppgaven ikke inneholder noen aktiviteter som krever risikovurdering. Dersom «JA»: Beskriv kort aktiviteteten i kartleggingskjemaet under. Risikovurdering trenger ikke å fylles ut.*

Signaturer: *Ansvarlig veileder:* Ole Ivar Sivertsen *Student:* Rasmus K. Skaare

ID nr.	Aktivitet/prosess	Ansvarlig	Eksisterende dokumentasjon	Eksisterende sikringstiltak	Lov, forskrift o.l.	Kommentar
1	Litteraturstudie og utvikling av AML-applikasjon	RKS				