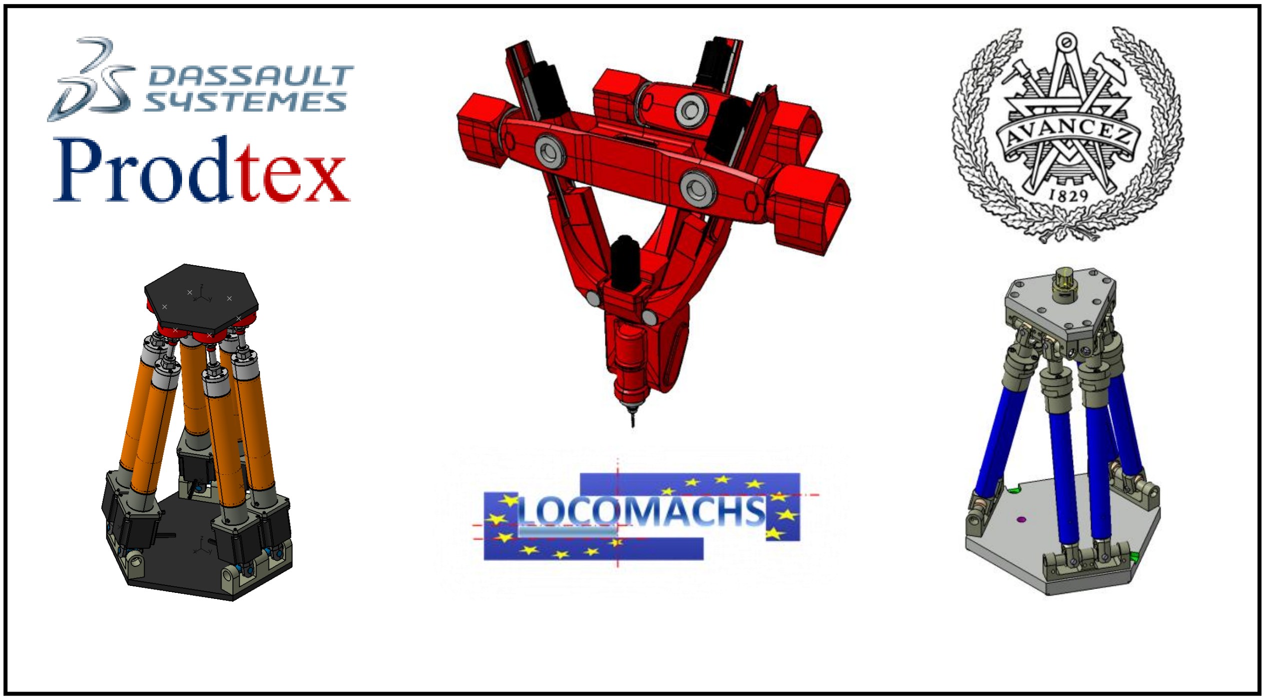
CHALMERS



Defining User-friendly Methodology to Create Advanced Mechanisms in DELMIA V5

Master's Thesis in the Production Engineering Master's Degree Program

ILKER ERDEM

Department of Product and Production Development
Master's Program in Production Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014

Defining User-friendly Methodology to Create Advanced Mechanisms in DELMIA V5

Master’s Thesis in the Production Engineering Master’s Program

ILKER ERDEM

Department of Product and Production Development
Master’s Program in Production Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014

Defining User-friendly Methodology to Create Advanced Mechanisms in DELMIA V5
Master's Thesis in Production Engineering
ILKER ERDEM

Master's thesis / Department of Product and Production Development
Chalmers University of Technology

Department of Product and Production Development
Division of Production Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone: + 46 (0)31-772 1000

ACKNOWLEDGEMENTS

In the accomplishment of this thesis work many have contributed, but the first and most important of all is my supervisor, Henrik Kihlman. I take this opportunity to thank him for his patience, generosity and strong belief in my efforts. Without his support and guidance, it would have been a very troublesome journey to embark and continue.

I also take this opportunity to express my gratitude to Peter Helgosson, a colleague of my supervisor and valuable member of Prodtex family, for his support in this thesis work. It has always been inspiring for me to see people who share the same passion for robots that I have carried in my all life. I, therefore, deeply thank Peter for his guidance, cordial support, important information and sincerity, which significantly helped me finish this assignment.

Another very important person who provided me the stepping stone is Torbjörn Jakobsson of Prodtex family. By his earlier work on hexapods and his efforts to create a kinematic device, I had been able to acquire the necessary knowledge and understand the dynamics behind a mechanism creation in DELMIA V5. Hence, I am very glad that I have been able to complete his work, without whom I believe this thesis work would have never been completed.

I am also obliged to say how grateful I am for the support and information that Mr. Fredrik Palmquist of Exechon family, Mr. Cyrille Froissart of Dassault Systemes, and Mr. Torgny Brogårdh of ABB provided. With their contributions, I have been able to move forward beyond the scopes of expectations.

Lastly, I would like to express countless thanks to my parents, my brother and friends for their constant encouragement and support without which success would not be achievable at all.

Defining User-friendly Methodology to Create Advanced Mechanisms in DELMIA V5
Master's Thesis in Production Engineering Master's Program

ILKER ERDEM
Department of Product and Production Development
Division of Production Engineering
Chalmers University of Technology

ABSTRACT

The main subject and aim of this thesis work are to create a methodology to define forward and inverse kinematics for advanced mechanisms such as robots with parallel and hybrid structures for the use in the project named LOCOMACHS (Low Cost Manufacturing and Assembly of Composite and Hybrid Structures) in DELMIA V5. These respective mechanisms are Hexapod/Flexapod 6P, Exechon and Gantry-Tau robots. The methodology created for this thesis work is building these robots with their forward and inverse kinematics definitions and testing the outcome. The forward kinematics aspect covers the building of the respective mechanisms whereas the inverse kinematics aspect includes the investigation of relevant theory and transfer of it to a C-file where this file is compiled to the use of DELMIA V5. Testing aspect focuses on comparing the results achieved with a C-file and MATLAB to the actual values coming from DELMIA V5. Hexapod and Flexapod 6P mechanisms are investigated and simulated in DELMIA V5 in complete structure while Exechon and Gantry-Tau robots are built only for their parallel structures.

Keywords: DELMIA V5, parallel kinematics, forward kinematics, inverse kinematics, Hexapod, Exechon, Gantry-TAU, simulation.

TABLE OF CONTENTS

DEFINITION OF TERMS AND ABBREVIATIONS

Actuator is a mechanism that initiates and control the motion of a system.

Degrees of freedom (DOF) is the term that describes the independent motions that a body is allowed to do.

CAD-model is a 2D/3D drawings or solid structure of parts in a modeling environment.

End-effector is the utility of a robot that interacts with the objects that are not part of the robot.

Joint is the center of motion where two solid structures of a robot contact each other.

Prismatic joint (P) is a joint type with one translational degree of freedom

Revolute joint (R) is a joint type with one rotational degree of freedom

Universal joint (U) is a two degrees of freedom joint that corresponds to two successively attached revolute joints.

Spherical joint (S) is a joint type with three degrees of freedom that allows parts to rotate in all axes freely.

Link/leg is the term that defines the solid structure between joints.

Kinematic chain is the term employed to describe the order of joints in a mechanism.

Serial kinematics machine (SKM) is a robot that the respective joints are successively attached to each other.

Parallel kinematic machine (PKM) is a machine that consists of links/legs that operate in parallel axes.

Hybrid robot/machine (HM) is a robot that consists of parallel and serial kinematic chains successively.

Tool Center Point (TCP) is the tip of the end-effector with respect to which the robot's interaction with environment is calculated.

1. INTRODUCTION

The EU financed project ©LOCOMACHS (Low Cost Manufacturing and Assembly of Composite and Hybrid Structures) is a joint adventure research and development program with SAAB AB as coordinator. The project is in partnership with 31 companies of aircraft industry and universities. LOCOMACHS mainly focuses on reduction or eradication of non-value adding operations in manufacturing of aircrafts. Thus, the objectives of the project can be summarized as

- Reduction of cost and lead-time
- Managing tolerance and variation
- Increasing the level of automation
- Designing new production methods and systems

At this point, this thesis work will fit in the last two objectives (LOCOMACHS 2014). In order to increase the level of automation and set up new rules for manufacturing and design, the simulation of respective ideas become vital. Thus, the necessary means to support new designing ideas must be defined in the chosen simulation environment – DELMIA V5.

DELMIA V5 of Dassault Systemes is one of the most advanced simulation tools that enables its users to create, define and control all aspects of a production system. As one of those aspects, DELMIA V5's *Device Building* module is a highly capable virtual mechanism creation instrument that defines forward and inverse kinematics of any given mechanism. However, advanced mechanisms such as parallel and hybrid (parallel-serial) robots are not recognized by this module directly via user-interface; thus, this definition of mechanism must be done by using C-code, which is considered to be a very important feature of DELMIA V5.

Hence with DELMIA V5's described feature, the means to support the newly designed manufacturing systems with higher levels of automation for LOCOMACHS project must be defined in terms of kinematic capabilities; and this thesis work is exactly formulated to provide a user-friendly methodology for the partners of project LOCOMACHS to simulate their kinematic devices in DELMIA V5.

1.1. OBJECTIVES AND SCOPE

In this thesis work, readers will find answers to how to define advanced mechanisms in DELMIA V5 by both using forward and inverse kinematics definition tools for robot types: hexapod/Flexapod 6P, Exechon and Gantry-Tau robots. So, the objective of this thesis work is to

- define a methodology to create advanced mechanisms in terms of forward kinematics
- create a methodic application of relevant theory for inverse kinematics to DELMIA V5
- continue the methodology to define how to set up the compilation and testing environment.
- simulate the respective robots by using the methodology.

Then, scope of this thesis work covers:

- Creation of methodology
- Building of declared robots in DELMIA V5 environment using the methodology
- Testing of the prospective robots in DELMIA environment

Therefore, the structure of this paper will be as following:

- Forward and inverse kinematics creation in DELMIA V5
- The analysis of C-files for inverse kinematics
- Theory for the given robots in terms of inverse kinematics and its transfer to a C-file
- Method employed during the building and solving the problems occurred
- The creation of Hexapod/Flexapod 6P, Exechon and Gantry-Tau robots in DELMIA V5 (appendix section)

1.2. METHODOLOGY

The scientific inquiry for this thesis work is originated on the question whether it is possible to formulate a methodology that enables a layman user to build advanced mechanisms in a simulation environment such as DELMIA V5 so that element of surprise in terms of unexpected errors is minimized. As Craig (2005) summarizes, the steps that any mechanism creation should cover assembly, kinematics definition and testing. In addition, the knowledge gained from the previous project owner – Torbjörn Jakobsson – and Cyrille Froissart of Dassault Systemes shows that the same way of thinking of Craig's applies to DELMIA V5 as well. However, the documented information from their work is either limited to serial kinematics machines or only defined within the limits of inverse kinematics for parallel kinematics machines. Thus, the remaining aspects of mechanism creation for parallel kinematics machines are still a black box. Therefore, the experimentation phase should be initiated before defining a user-friendly methodology. This kind of experimentation and thinking was also conducted by Torbjörn Jakobsson from which he was also able to gain great amount of insight regarding advanced mechanism creation in DELMIA V5. By focusing on hexapod robot creation in terms of inverse kinematics definition, he was able to map all the important founding dynamics for inverse kinematics creation and documented it. Thus, it can be hypothesized that by assembling the respective robots for all the steps of mechanism creation it would be possible to record a map of actions from which a user-friendly methodology can be developed.

The results of this experimentation and the map of actions that create the methodology is given with details in Chapter 5: RECOMMENDED COURSE OF ACTION FOR MECHANISM BUILDING IN DELMIA V5.

2. INTRODUCTION TO ROBOTS: HEXAPOD/FLEXAPOD 6P, EXECHON AND GANTRY-TAU

In this chapter, a small introduction will be given to respective robots of this thesis work along with their importance in industry.

2.1. HEXAPOD/FLEXAPOD 6P

Hexapod and Flexapod 6P are the same class of robots, where the only difference lies in their design and the structure of the parts used. This difference of design does not affect the kinematics or the idea behind this class. Thus, these robots in this chapter will be described schematically.

Hexapod robots, also known as Stewart Platforms, named after the creator D. Stewart, is parallel structure of two platforms connected to each other via prismatic actuators named as legs. As the name suggests these six legs are connected to a base platform in one end whereas the other ends of the legs are attached to a mobile platform (Yang 1998). The design difference between Flexapod 6P and a standard hexapod is that upper attachment points for hexapod are connected to legs via spherical joints while Flexapod 6P has universal joints for upper attachment points and the rotation of legs about their own axis provides the final degree of freedom that a spherical joint grants in a hexapod.

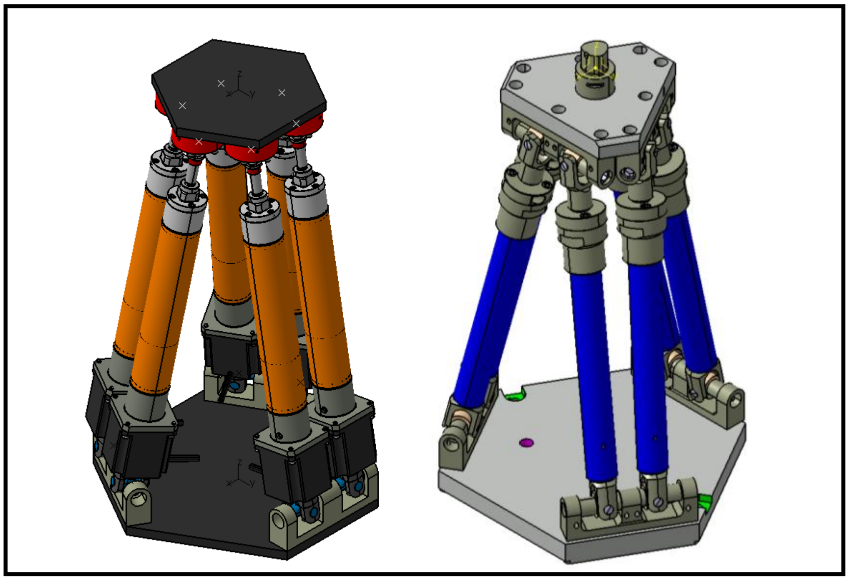


Figure 1: Hexapod (on the left) and Flexapod 6P (on the right)

This class of robots that is going to be adapted to DELMIA V5 will be utilized in the project called LOCOMACHS in which hexapods will be used as flexible tooling equipment to support and enhance the quality of production in aerospace industry.

2.2. EXECHON

One of the most successful PKM (Parallel Kinematic Machine) structures Exechon – developed and patented by Karl-Erik Neumann in 2006 – is a tricept-related hybrid machine where a parallel structure of three legs actuated with prismatic joints is followed by an R-R or a spherical wrist. The structure itself was created in Sweden and is currently being manufactured by several companies in the world (Zoppi 2010).

The kinematics structure is comprised of parallel and serial parts which are attached successively and creating the hybrid kinematics. The parallel structure has three legs as said above and two of which are identical to each other. These two legs create a common plane meanwhile the third leg's plane is perpendicular to the plane created by identical legs. The joints for the identical legs follow RR-P-R and those for the third leg are RRR-P-R (Bi 2010). This is illustrated in figure 2.

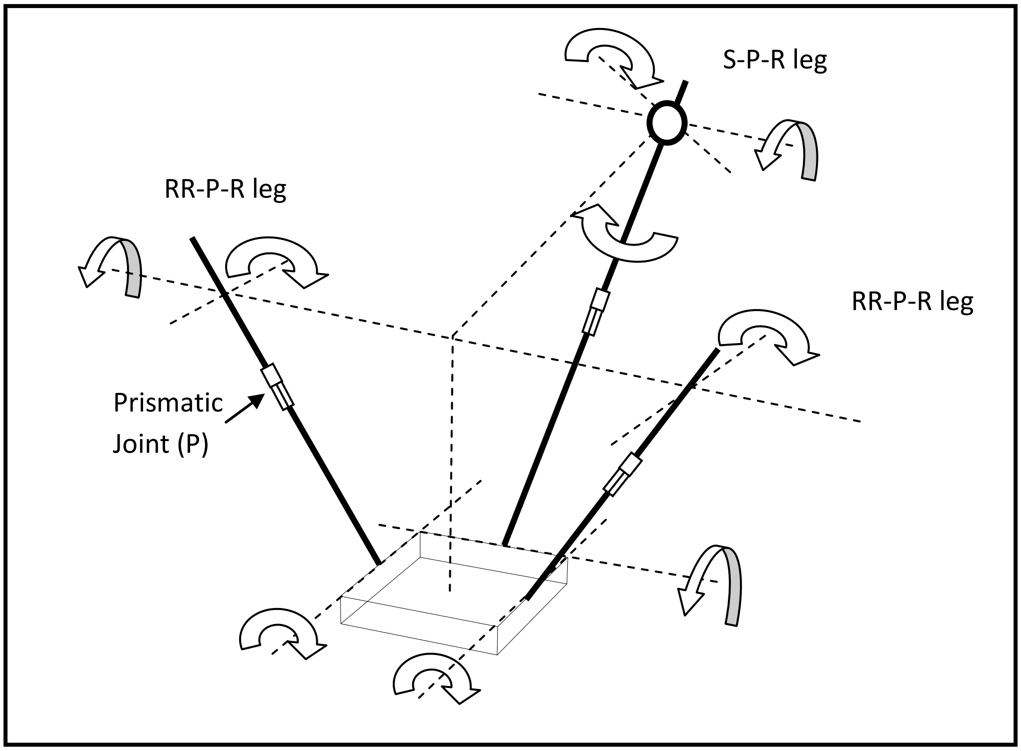


Figure 2: Exechon's parallel structure

The successively attached serial structure can be a revolute-revolute or spherical wrist. The tool attached to wrist can be used for drilling or milling operations as Bi (2010) states. The complete structure built for DELMIA V5 can be seen in figure 3 .

In this thesis work, revolute-revolute wrist is not going to be utilized; and thus, the calculations will be made only for the parallel structure.

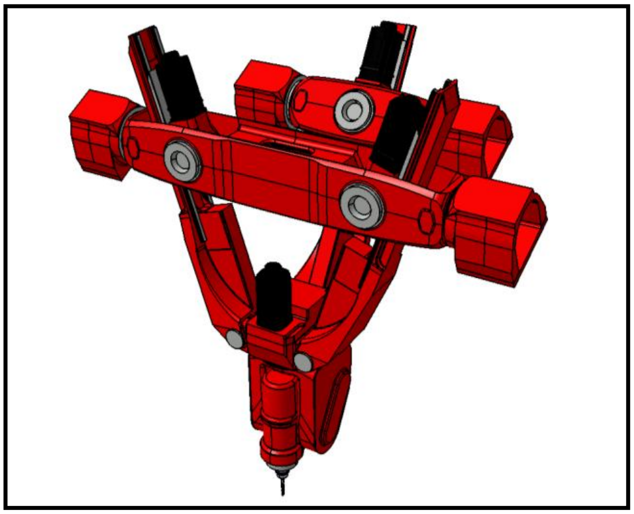


Figure 3: Exechon robot as CAD model in DELMIA V5

2.3. GANTRY-TAU

Gantry-Tau is a parallel kinematics robot patented by ABB. The parallel structure consists of three clusters in which links are attached to mobile platform on different points. The reason that Gantry-Tau robot has gantry term is the fact that the actuated motion provided by three prismatic joints have the same ideology with Cartesian (also known as Gantry) robots. The clusters term used here refer to the group of links where each group connects a prismatic joint to the end-effector. In total, the structure has six links and clustered as 3-2-1. This notation describes the number of links that each cluster has. The kinematic chain of each link is a PRRS and the actuation is in the prismatic joint. The chains and their relations to other parts can be seen in figure 4. (Johannesson 2003)

On the other hand, it is important to keep in mind that in practical applications the kinematic chain can be changed to a PSS (Prismatic-Spherical-Spherical). The reason is that the extra DOF that comes with the first spherical joint only provides rotation about link’s own axial axis. Thus, in practice, this has no influence. However, when simulated such extra joint will cause problems; and therefore, the regular PRRS chain should be used as shown in figure 4.

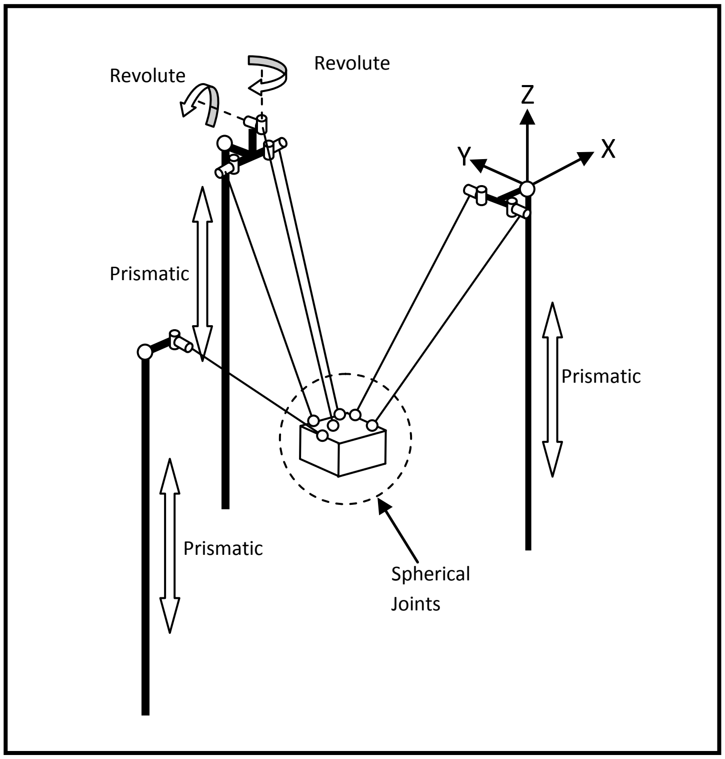


Figure 4: Gantry-Tau robot

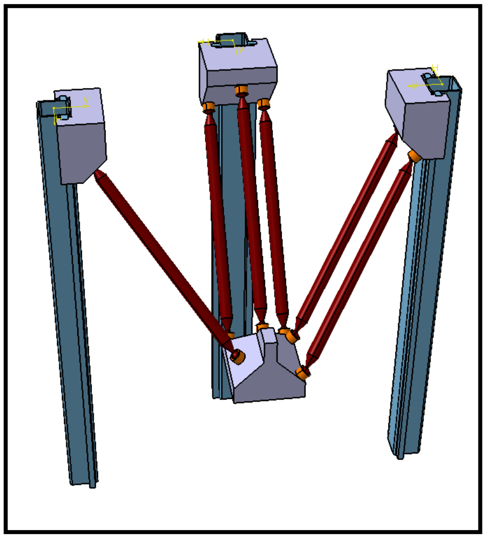


Figure 5: Gantry-Tau model in DELMIA V5

3. THEORY

In this chapter, forward/inverse kinematics in general will be described along with their uses in DELMIA. Also, the relevant theory for the inverse kinematics of Hexapod/Flexapod 6P, Exechon and Gantry-Tau robots will be presented along with their respective C-files.

3.1. FORWARD KINEMATICS

Before going deep into detail, it is important to define the term, kinematics. Kinematics, then, is the investigation of mechanisms in terms of position, velocities and accelerations without including the forces that set the basis for them (Craig 2005).

This kinematics definition, by theory, is divided into two categories where the first analysis method is called forward kinematics. This analysis can be defined as attaching coordinate frames to each link in a robot until the end-effector (TCP) in order to describe the position and orientation of the end-effector in terms of base-coordinate system (Jazar 2010). These terms are shown in the figure 6.

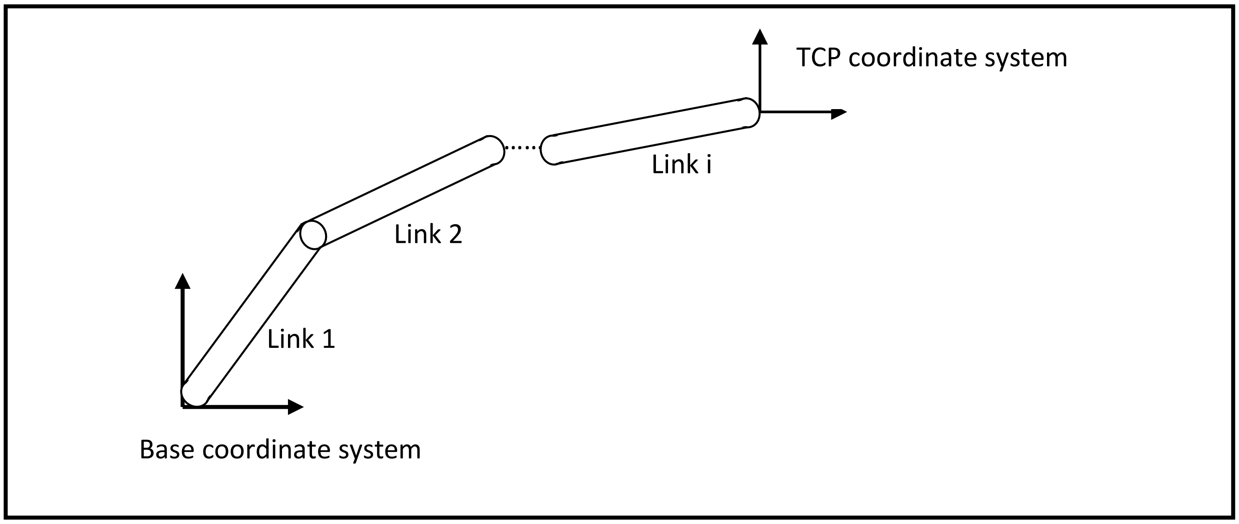


Figure 6: Basic terms in kinematics

Thus; the idea in forward kinematics is to describe the given TCP coordinate system in base-coordinate system by attaching intermediary coordinate frames at every joint. Hence, by propagating from one coordinate frame to the next the end-effector’s orientation and position is described by using each link’s coordinate frame variables where each variable is chosen as a joint value in which actuation by command is possible. However, in order to propagate from one frame to another, a means that includes relevant information about a frame is necessary. This relevant information should include the position of the origin point and the orientation of the consisting XYZ unit vectors with respect to another frame. In kinematics theory, the means to convey this information is achieved with a 4x4 matrix, named *Transformation matrix* (Craig 2005).

To fully understand and describe what transformation operation is, three different coordinate frames A, B and C are shown in figure 7 along with point C_P. To describe this point C_P in coordinate frame A, transformation matrices between A-B and B-C must be created.

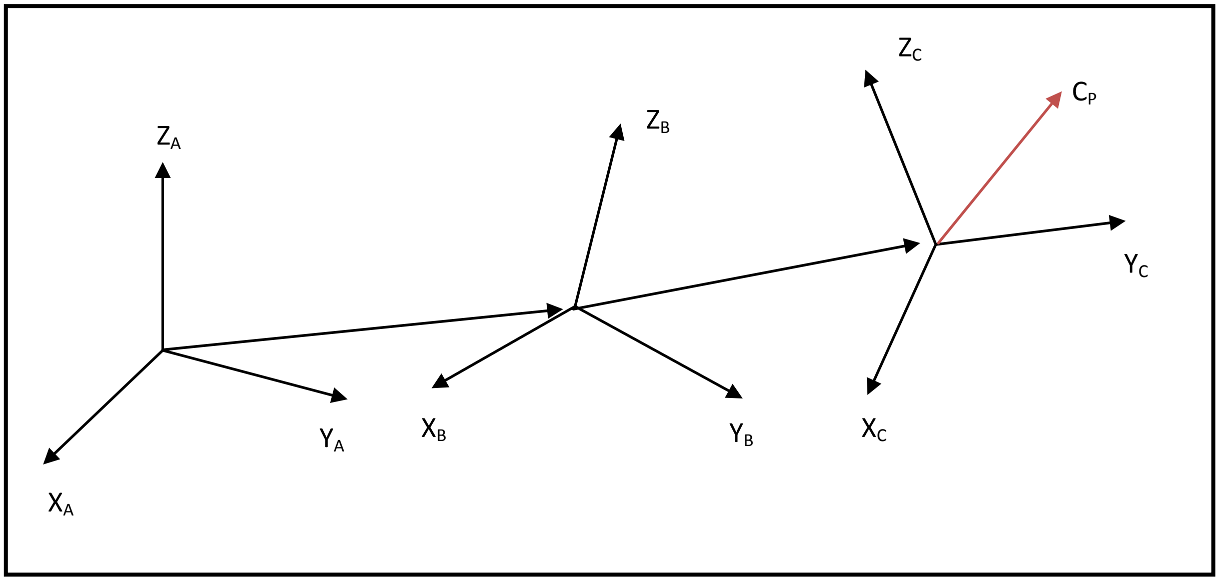


Figure 7: A, B and C coordinate frames each relative to earlier one.

A transformation matrix then is composed of rotation and translation and has the following structure.

$$ {}^{A}_{C}T = \left[\begin{array}{c|c} \text{Rotation} & \text{Translation} \\ \hline 0 \quad 0 \quad 0 & 1 \end{array}\right] \tag{1.1}$$

${}^{A}_{C}T$ notation describes a transformation between A and C coordinate frames in which C is relative to A-coordinate frame. Thus, when multiplied with ${}^{A}_{C}T$, any given point or vector in C-coordinate frame is transformed or described in coordinate frame A (Craig 2005). To create this transformation matrix by using all the given coordinate frames, the following formula is applied.

$$ {}^{A}_{C}T = {}^{A}_{B}T \; {}^{B}_{C}T \tag{1.2}$$

This formula states that {C} is first described relative to {B}, and then multiplied with the transformation between {A} and {B} in which {B} is relative to {A}. Hence, transformation between {A} and {C} is completed by using an intermediary coordinate frame {B} (Craig 2005).

The terms rotation and translation in ${}^{A}_{C}T$ are the compounds of this matrix where rotation is the unit vector definition of each axis of {C} relative to {A} and translation is the vector between the origin points of {A} and {C} relative to {A}. The last row of this transformation matrix is [0, 0, 0, 1] and this row has no significance but is only useful in terms of making the matrix square (Craig 2005).

Hence, the vector C_P is multiplied with the transformation matrix ${}_{C}^{A}T$ and the outcome of this operation is the vector defined in {A}. This operation is formulated as

$$ {}^{A}C_{P} = {}_{B}^{A}T \; {}_{C}^{B}T \; {}^{C}C_{P} \tag{1.3} $$

When the formula 1.3 is applied, a robot's end-effector is described through each link's transformation and the resulting transformation matrix includes variables for the actuators in the robot mechanism. When these variables entered, this transformation matrix yields the position and orientation of end-effector or in other words TCP location and orientation (Craig 2005).

This methodology of course can be applied to a parallel structure such as hexapods. On the other hand, the mathematical complexity becomes a great burden and solving these mathematical expressions may not yield an exact result but sometimes estimations due to the necessity for numerical approaches (Yang 1998). Therefore, DELMIA at this point provides a very useful way to create a mechanism and builds the forward kinematics description automatically at the end-effector.

3.2. BUILDING MECHANISMS AND FORWARD KINEMATICS IN DELMIA V5

In this chapter how DELMIA V5 approaches the concept of forward kinematics will be described whereas building the complete mechanisms along with their forward kinematics description will be given in appendix for hexapod/Flexapod 6P, Exechon and Gantry-Tau robots. The information presented in this chapter is based on Cyrille Froissart's confidential documentation. Thus, due to confidentiality of the document the reference cannot be given.

3.2.1. CREATION OF MECHANISMS IN DELMIA V5

Creation of a mechanism can be achieved in three ways in DELMIA environment. These methods are

- Classic method
- Frame of Interest Method (FOI)
- Frame of Interest and Dress-Up Method

In this thesis work only "Frame of Interest Method" is used since it offered simplicity and geometry-independent mechanism creation. FOI method is used by appointing frames (in this case called as Frame of Interest) to the regarding joint locations and creating pre-defined joints from these frames. These pre-defined joints are

- Revolute (1 rotational degree of freedom)
- Prismatic (1 translational degree of freedom)
- Universal (2 rotational degrees of freedom)
- Cylindrical (1 rotational or translational degree of freedom)
- Spherical (3 rotational degrees of freedom)

The creation of a mechanism in DELMIA V5 starts with opening '*Device building'* module. In this module, the user first creates a mechanism in the node. Afterwards, user defines a fixed part and starts to build the respective mechanism upon that fixed part. To simply illustrate, an example will be given by embarking the FOI method.

In this example, one leg of a Flexapod 6P will be used along with the fixed part '*Base'*. First, DELMIA V5 will be opened in '*Device building'* module.

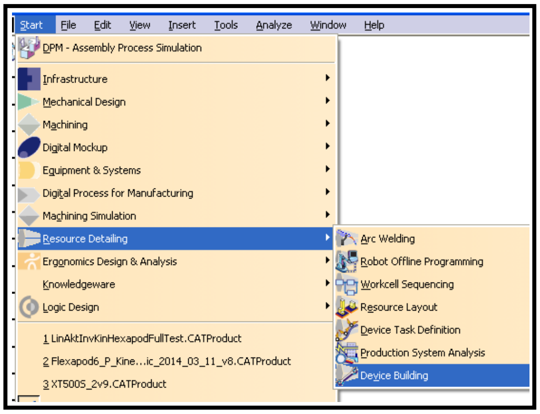


Figure 8: Device Building module in DELMIA V5

In the second step, for each part of the mechanism *'new component'* is clicked and a separate component is created under the node tree. For this example, necessary parts are

- Base
- Lower connecting cube
- Lower leg connected to base
- Upper leg that is connected to lower leg part by a prismatic joints

Thus, respective node tree will be looking as in figure 9.

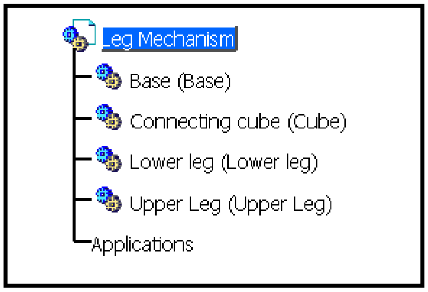
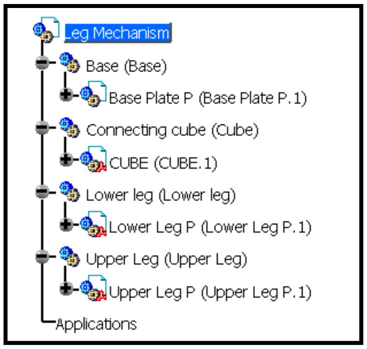


Figure 9: Example of a node tree for mechanisms

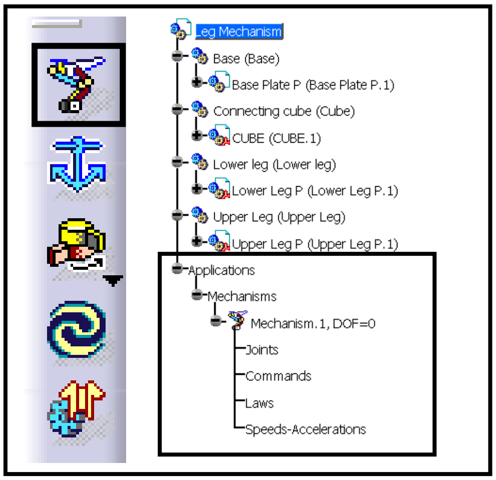
In the third step, necessary CAD-models should be inserted into the respective nodes by using '*Insert-Existing Component*' commands. Then, the node tree should have the form in figure 10.



Leg Mechanism
Base (Base)
Base Plate P (Base Plate P.1)
Connecting cube (Cube)
CUBE (CUBE.1)
Lower leg (Lower leg)
Lower Leg P (Lower Leg P.1)
Upper Leg (Upper Leg)
Upper Leg P (Upper Leg P.1)
Applications

Figure 10: Leg mechanism with CAD-models inserted

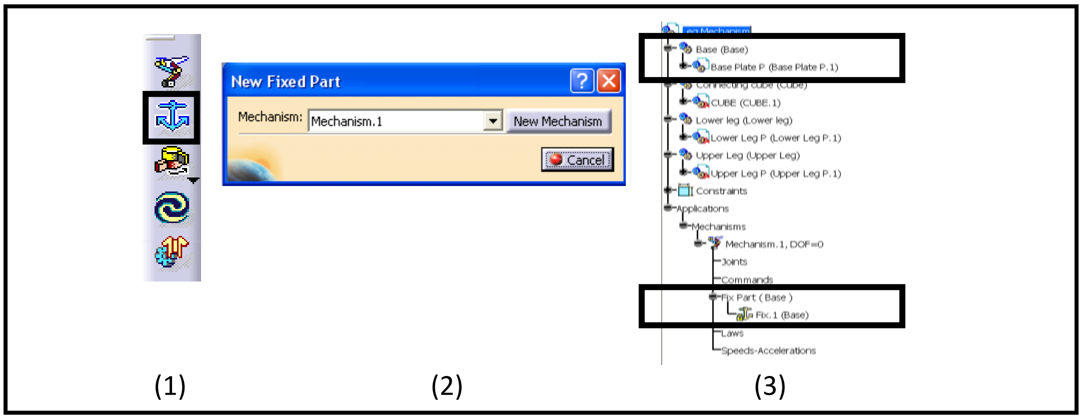
In the fourth step, a new mechanism will be created and a fixed part –which in this case is the part called *'Base'*- will be appointed. To achieve this, first *'New Mechanism'* button must be clicked. Then in the node tree under *'Applications'* a new mechanism division will be available.



Leg Mechanism
Base (Base)
Base Plate P (Base Plate P.1)
Connecting cube (Cube)
CUBE (CUBE.1)
Lower leg (Lower leg)
Lower Leg P (Lower Leg P.1)
Upper Leg (Upper Leg)
Upper Leg P (Upper Leg P.1)
Applications
Mechanisms
Mechanism.1, DOF=0
Joints
Commands
Laws
Speeds-Accelerations

Figure 11: Creation of a new mechanism

To appoint the fixed part, click the 'Fixed Part' and a respective menu will appear. Afterwards, respective part *'Base'* must be chosen for this example.



New Fixed Part
Mechanism: Mechanism.1

Leg Mechanism
Base (Base)
Base Plate P (Base Plate P.1)
Connecting cube (Cube)
CUBE (CUBE.1)
Lower leg (Lower leg)
Lower Leg P (Lower Leg P.1)
Upper Leg (Upper Leg)
Upper Leg P (Upper Leg P.1)
Constraints
Applications
Mechanisms
Mechanism.1, DOF=0
Joints
Commands
Fix Part (Base)
Fix.1 (Base)
Laws
Speeds-Accelerations

Figure 12: Creating the fixed part

In the fifth step, first joint will be created between the cube and the base part. Since the base part is designed to have six legs, in this example only one leg will be demonstrated. For the remaining legs, refer to the appendix.

To create a revolute joint, first Frames of Interest (FOIs) will be attached to the respective locations. The first frame will be attached to the base and the second in the cube. The order does not matter, but it is noted that in order to avoid confusion, it is important to follow a pattern. At first, click *'Frames of Interest'* button and create a Frames of Interest node under both parts – *'Cube'* and *'Base'*. Then, click on 'Frame type' button and place the FOI as shown in figure 13.

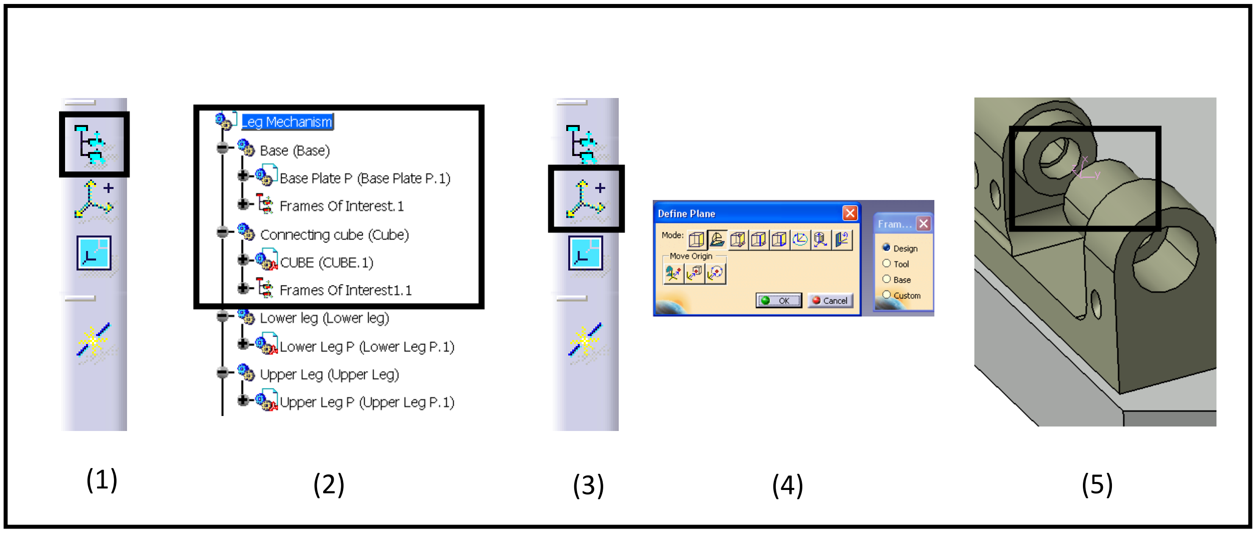


Figure 13: Appointing FOI to *'Base'* part

Then the same procedure will be applied to cube and the result is shown in figure 14.

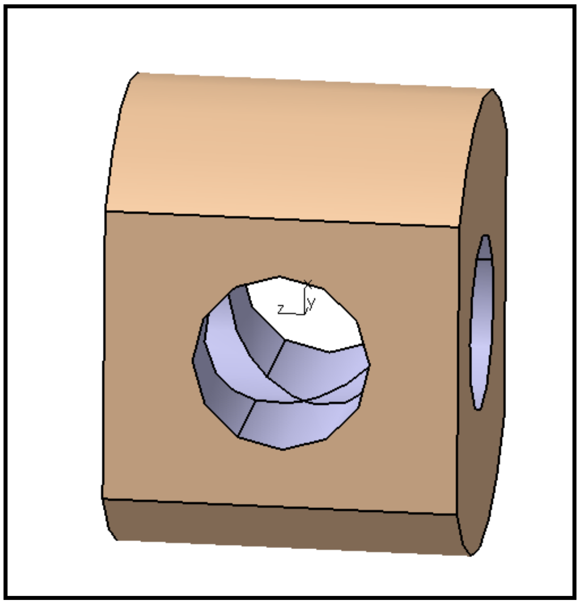


Figure 14: Cube part with FOI at the center

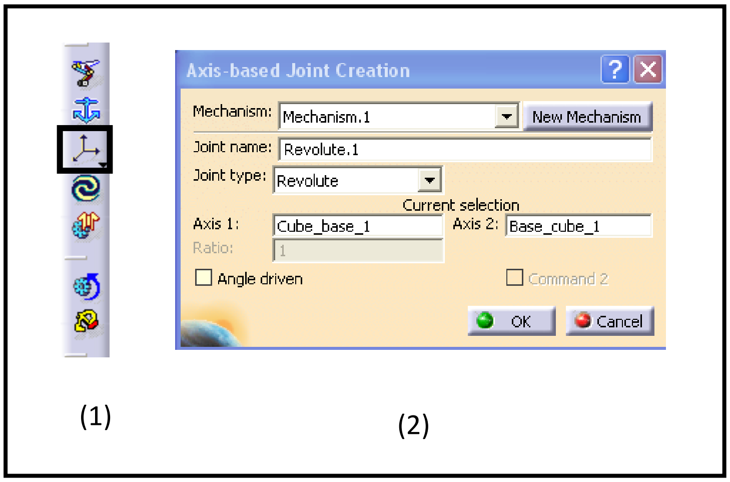
It is important point out that all one-degree-of-freedom joints either translate or rotate about the Z-axis of the FOI. Thus, as seen in figure 13 and figure 14, the rotational axis Z is about to coincide when the joint is created.

Since all the necessary FOIs are attached, the revolute joint now can be created. To create a joint, DELMIA V5 offers various ways. However, as stated before, when FOIs are used creation of joints are easily done by using *'Joint from axis'*.

After opening *'Joint from axis',* a new menu appears where users define the properties of the axis to be created. These features are

- Mechanism to which the joint will belong
- Joint name
- Joint type
- Axes required for joint creation.

In this case, these features will be Mechanism.1, Revolute and 2 FOIs created for 'Base' and 'Cube'. This is shown in figure 15 .


Axis-based Joint Creation

Mechanism: Mechanism.1

Joint name: Revolute.1

Joint type: Revolute

Current selection

Axis 1: Cube_base_1 Axis 2: Base_cube_1

Ratio: 1

□ Angle driven □ Command 2

Figure 15: Joint creation with FOIs

The order of the joints does not matter in this case and *'Joint name'* section is automatically generated. The *'Angle driven'* button makes the joint an actuator in the mechanism. When clicked *'OK'*, the cube is automatically attached to the correct location where two FOIs coincide. The result is shown in figure 16.

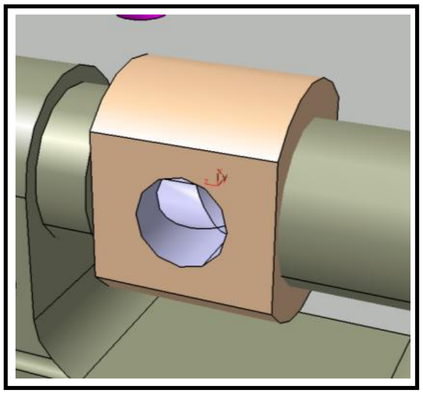


Figure 16: Revolute joint with FOIs

Second part of this mechanism is to connect *'Lower Leg'* part to the cube with a revolute joint. Again, the same procedure will be followed as for the joint between the base and cube except this time one of the FOIs will be attached to *'Lower Leg'*. The FOIs can be seen in figure 17.

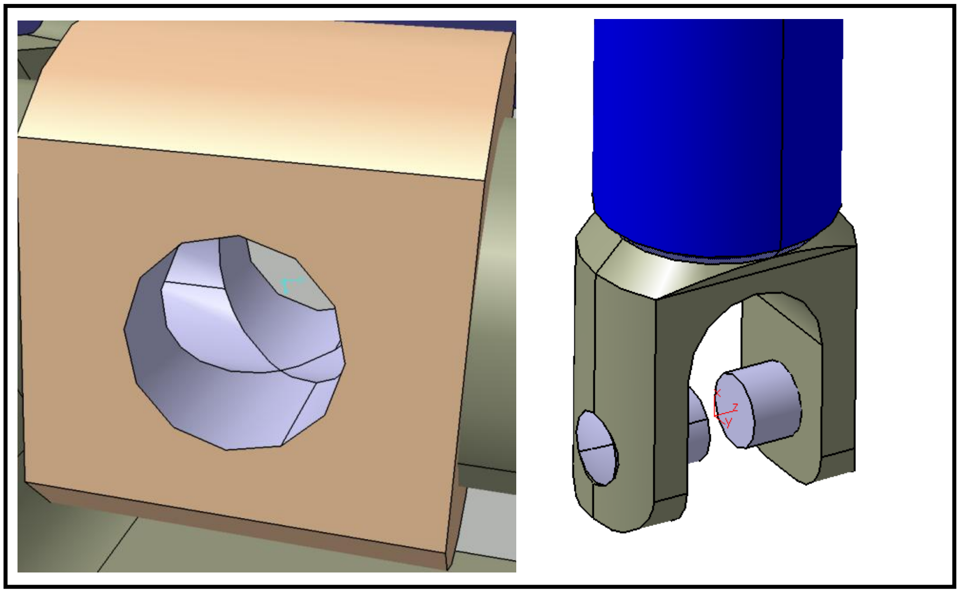


Figure 17: FOIs for the second revolute joint

When these two FOIs are combined by using the same methodology for the first revolute joint, the lower leg is automatically translated to the position and oriented in a way where two FOIs' Z-axes coincide. The result is shown in figure 18.

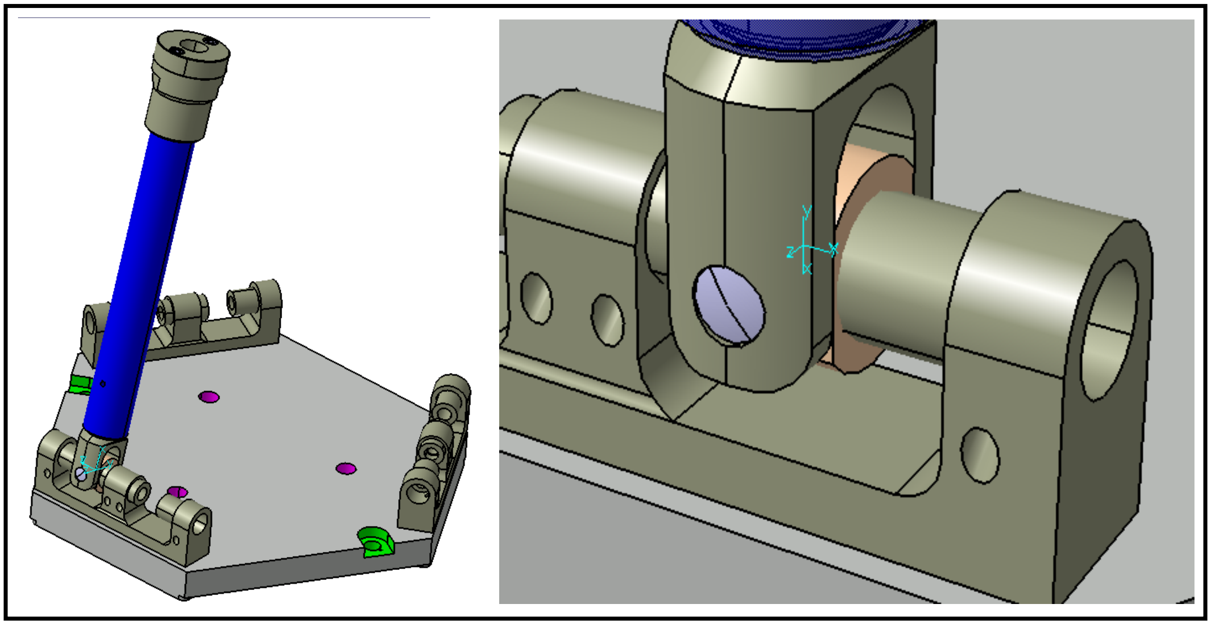
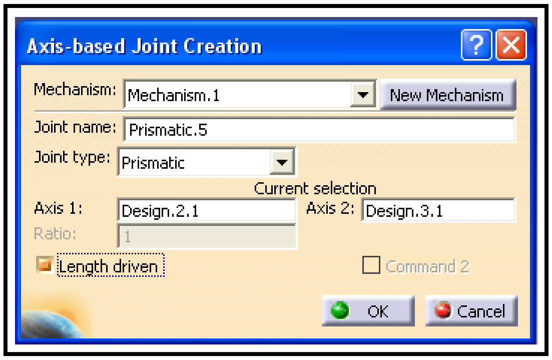


Figure 18: The second revolute joint

In the third step, a prismatic joint will be created between *'Lower Leg'* and *'Upper Leg'* parts. To achieve a correct state with the joint, it is important to place the FOIs in the accurate positions. Specifically, when a prismatic joint is created, DELMIA V5 takes the current positions of each FOI and makes them coincide in Z-axes. However, since the origin points are not necessarily coincided for a prismatic joint DELMIA V5 sets the zero position of this joint in the same coordinates where the respective parts currently are. Therefore, it is important to have each FOI at the same location before a prismatic joint is created. To achieve this, one may consider creating a revolute joint and then deleting it along with the constraints; and afterwards, creating the revolute joint. The second way is to move the mobile part from respective FOI to the coordinates of the one of fixed part in the joint.

With the first way chosen, a revolute joint will be created with the same routine for earlier joints. Then revolute joint will be deleted and from the *'Joint from axes'* button, a prismatic joint will be directly made. Since this joint is appointed as the actuator of the mechanism, the *'Length driven'* command will be activated. The respective FOIs and the joint creation menu are shown in figure 19 and figure 20.



Axis-based Joint Creation

Mechanism: Mechanism.1 | New Mechanism

Joint name: Prismatic.5

Joint type: Prismatic

Current selection

Axis 1: Design.2.1 Axis 2: Design.3.1

Ratio: 1

☑ Length driven ☐ Command 2

OK Cancel

Figure 19: Prismatic Joint menu with *'Length Driven'* activated

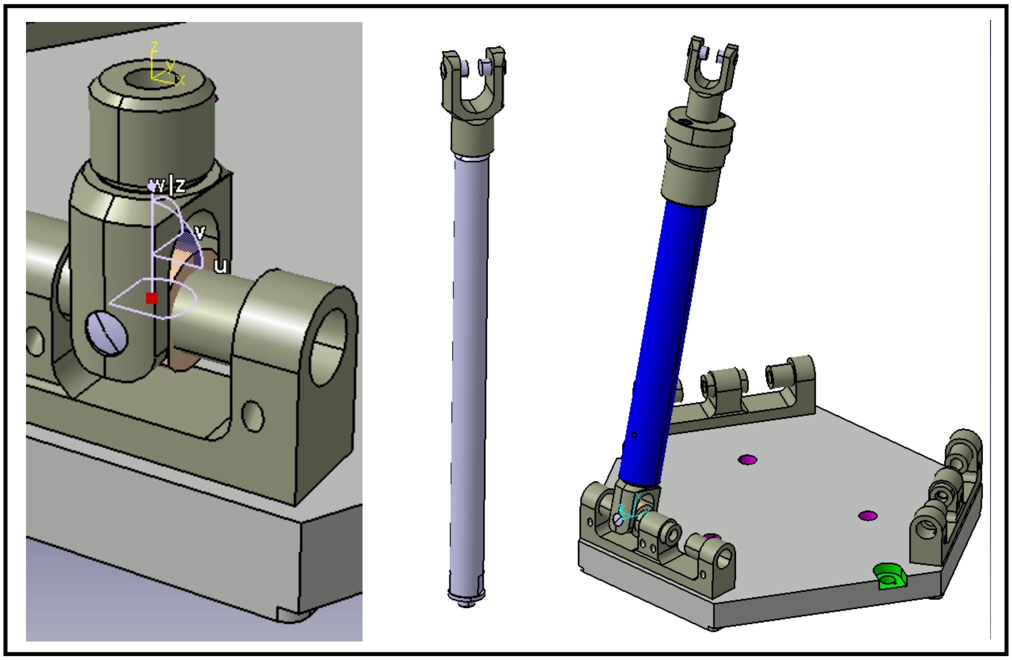
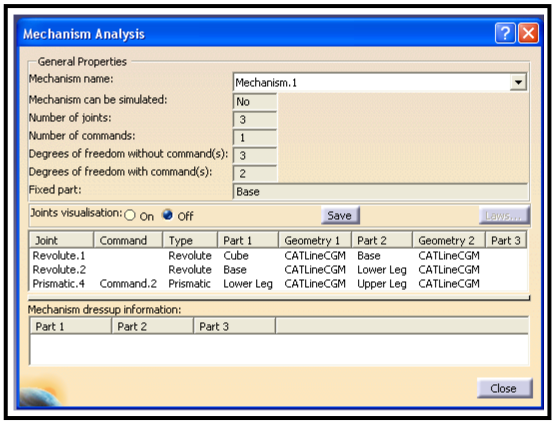


Figure 20: Prismatic Joint and respective FOIs

As seen earlier, the revolute joints are not created as actuators; therefore, DELMIA V5 will not be able to simulate the system due to free motion of those joints. However, since this is an illustration to show how mechanism creation works, it would be possible to test the mechanism by making all the joints actuators. This achieved via double-clicking on each revolute joint in mechanism node and activating *'Angle driven'* field. This way DELMIA V5 will prompt a menu on which the following information can be seen *'The mechanism can be simulated'*. This feature can also be observed by clicking the mechanism properties ⚙ icon. On the prompted menu, DELMIA V5 displays all the joints created and their respective parts. In this section, a very important aspect is also shown in which the total degrees of freedom of a mechanism can be seen. These degrees of freedom are divided into two sections where one shows *Degrees of freedom with command* and the other *Degrees of freedom without command*. The *Mechanism Analysis* menu is shown in figure 21.



Mechanism Analysis

General Properties

Mechanism name:	Mechanism.1
Mechanism can be simulated:	No
Number of joints:	3
Number of commands:	1
Degrees of freedom without command(s):	3
Degrees of freedom with command(s):	2
Fixed part:	Base

Joints visualisation: ○ On ● Off

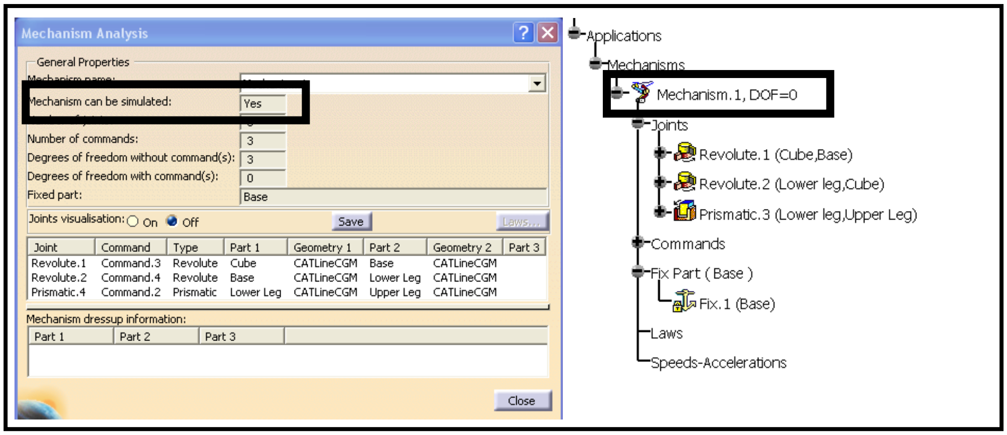
Joint	Command	Type	Part 1	Geometry 1	Part 2	Geometry 2	Part 3
Revolute.1		Revolute	Cube	CATLineCGM	Base	CATLineCGM	
Revolute.2		Revolute	Base	CATLineCGM	Lower Leg	CATLineCGM	
Prismatic.4	Command.2	Prismatic	Lower Leg	CATLineCGM	Upper Leg	CATLineCGM	

Mechanism dressup information:

Part 1	Part 2	Part 3

Figure 21: Mechanism properties menu

In figure 21, it is seen that the mechanism has three joints and only one of them has a command, in other words one of them is only actuated. This means that the rest of the joints are free to respond any action which makes them dangling joints by the terminology employed by DELMIA V5. This does not mean that the mechanism created is incorrect but unfinished. As said earlier, this is only a demonstration and the complete structure will be presented in appendix section of this paper. Thus, in order to simulate the system, the revolute joints will be made actuators. The user must click on each joint created under the *'Mechanism-Joints'* node in the product tree and double-click on each joint. The same menu for joint creation will be prompted and on that menu, *'Angle driven'* field should be activated. When repeated for the other revolute joint as well, the mechanism properties menu will display '*Yes*' for the section '*The mechanism can be simulated*'. Also, it will be seen that degrees of freedom without command will be zero. This can also be seen from the mechanism node in the product tree as shown in figure 22.



Mechanism Analysis

General Properties

Mechanism name:

Mechanism can be simulated: Yes

Number of commands: 3

Degrees of freedom without command(s): 3

Degrees of freedom with command(s): 0

Fixed part: Base

Joints visualisation: ○ On ● Off

Joint	Command	Type	Part 1	Geometry 1	Part 2	Geometry 2	Part 3
Revolute.1	Command.3	Revolute	Cube	CATLineCGM	Base	CATLineCGM	
Revolute.2	Command.4	Revolute	Base	CATLineCGM	Lower Leg	CATLineCGM	
Prismatic.4	Command.2	Prismatic	Lower Leg	CATLineCGM	Upper Leg	CATLineCGM	

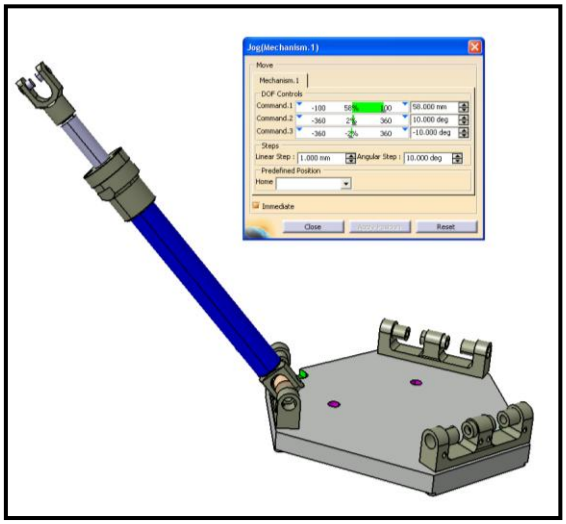
Mechanism dressup information:

Part 1	Part 2	Part 3

Applications
- Mechanisms
  - Mechanism.1, DOF=0
    - Joints
      - Revolute.1 (Cube,Base)
      - Revolute.2 (Lower leg,Cube)
      - Prismatic.3 (Lower leg,Upper Leg)
    - Commands
    - Fix Part (Base)
      - Fix.1 (Base)
    - Laws
    - Speeds-Accelerations

Figure 22: Mechanism properties after actuation

To see the behavior of the mechanism, it must be jogged. This property is provided by 'Jog mechanism' button . When clicked, a new menu will prompt, and on this menu the user will be able to jog the mechanism for given range of joints. The menu and the jogged mechanism can be seen in figure 23.



Jog(Mechanism.1)

Move

Mechanism.1

DOF Controls

Command.1	-100	58%	100	58.000 mm
Command.2	-360	2%	360	10.000 deg
Command.3	-360	-2%	360	-10.000 deg

Steps

Linear Step : 1.000 mm Angular Step : 10.000 deg

Predefined Position

Home

□ Immediate

Figure 23: Jogging the mechanism

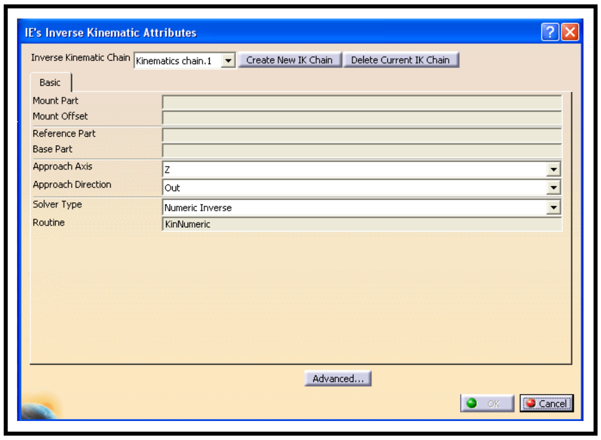
This way DELMIA achieves the forward kinematics of any model needed to be built. It requires no other calculation; and the required transformation matrix is automatically created when the inverse kinematics definition is made – which will be the topic of the next section.

3.3. INVERSE KINEMATICS

Inverse kinematics is the way of finding necessary joint values for a given TCP values of a robot. The way of reaching a solution is divided into two methods, algebraic and geometric. Algebraic solution is based on finding joint values by acquiring equations from transformation matrix with given values of TCP. On the other hand, geometric solution is about decomposing the spatial definition of a robot into several planar equations by using vector definitions (Craig 2005). In the case of this paper, the hexapod (Flexapod 6P) uses a geometric solution whereas Exechon and Gantry-Tau robot utilizes both of the given ways. The building of inverse kinematics in DELMIA V5 will be the topic of this section whereas the inverse kinematics theory of the respective robots will be presented in the next chapter.

3.3.1. CREATION OF INVERSE KINEMATICS IN DELMIA V5

The inverse kinematics definition in DELMIA V5 is done by the *'Inverse Kinematics'* icon . When this icon is clicked and the mechanism is chosen, DELMIA V5 prompts a new menu for the definition of inverse kinematics. The menu is show in figure 24.


IE's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

Basic

Mount Part:

Mount Offset:

Reference Part:

Base Part:

Approach Axis: Z

Approach Direction: Out

Solver Type: Numeric Inverse

Routine: KinNumeric

Figure 24: Inverse kinematics definition menu

The first tab in the menu is *Basic*. *Mount part* section is used to define the part that is attached to the TCP. *Mount offset* is used for the FOI that describes the TCP's coordinate frame. *Reference part* is used to define the coordinate frame that is going to be the reference for calculations where as the *Base part* is the fixed part of the mechanism. In many cases *Base part* and *Reference part* are the same.

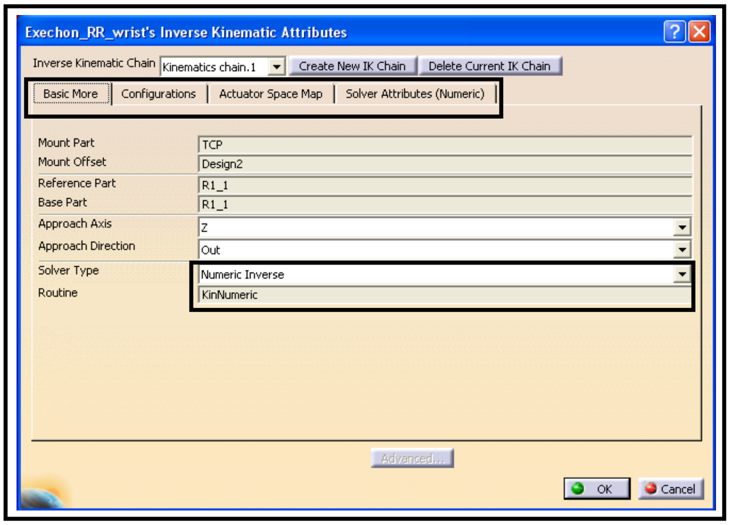
Approach axis is used to define the main axis of the TCP which in most of the cases is Z-axis whereas *Approach direction* is the direction that defines the positive direction in calculations whether it is inwards or outwards on *Approach axis*. It is usually set as *Out.*

The solver type provides options to define the inverse kinematics solution. These options are

- Numeric method
- Generic method
- Device-specific method
- User inverse method (use of a C-code).

In this paper, the creation of inverse kinematics will be carried out by creating a C-code. On the other hand, it is important to represent how other methods work as well. Thus, simple instructions will be given for them.

Numeric method is a built-in feature of DELMIA V5. In this method, solver tries to solve the joint values by using algebraic methods from transformation matrix. The user has no chance to interfere with the calculation but define which joints to be solved. When this method is chosen and *'Advanced'* button is clicked, a new set of tabs appear in the inverse kinematics menu – which can be seen in figure 25.



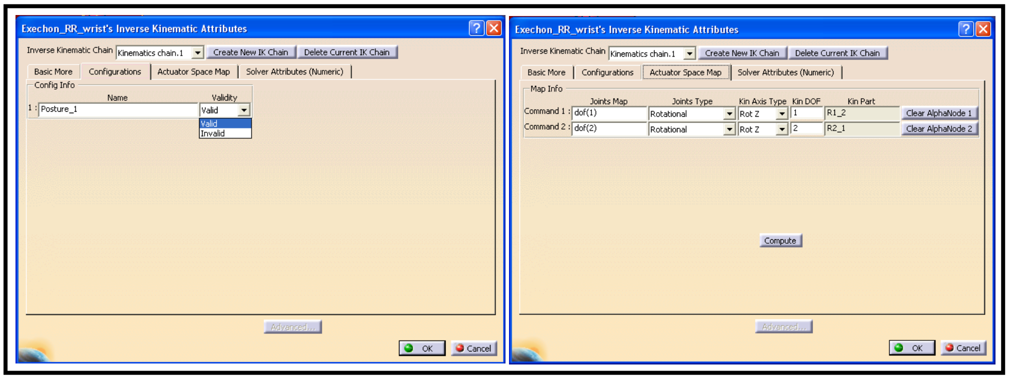
Exechon_RR_wrist's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

Mount Part	TCP
Mount Offset	Design2
Reference Part	R1_1
Base Part	R1_1
Approach Axis	Z
Approach Direction	Out
Solver Type	Numeric Inverse
Routine	KinNumeric

Figure 25: Inverse kinematics tabs for numeric method

In ‘*Configurations*’ tab, DELMIA V5 asks user to flag the postures of the given robot. These postures are flagged as valid or invalid where valid makes the posture of the robot available in the simulation environment. ‘*Actuator Space Map*’ tab is where the user maps the joints and their mobile parts with the commands for inverse kinematics calculation. In this tab, the user defines the joint map section first where each ‘Degree of Freedom’ is associated with the corresponding joint. For example, for RR mechanism of Exechon’s wrist (revolute and revolute in serial order) *dof(1)* appoints the first degree of freedom to revolute joint 1 whereas *dof(2)* appoints the second degree of freedom to revolute joint 2. In the second section of mapping, types of freedom are defined where the options are limited to translational or rotational. In the third section – *Kin Axis Type*, the main axis of motion is defined. For *Kin Part*, the mobile part of the joint is appointed. *Compute* button is used to appoint these values automatically, and it is possible that DELMIA V5 may not guess the entire system correctly. All these sections can be seen in figure 26.



Exechon_RR_wrist's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

Config Info

	Name	Validity
1 :	Posture_1	Valid

Exechon_RR_wrist's Inverse Kinematic Attributes

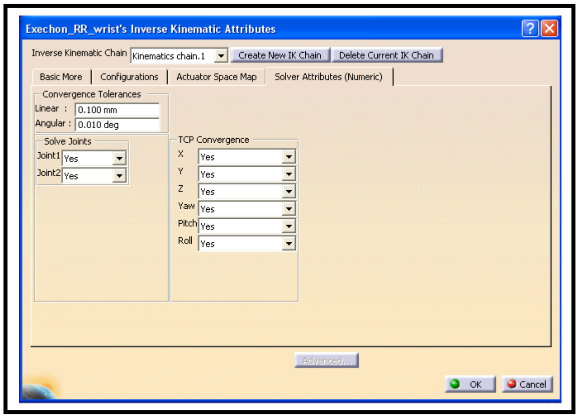
Inverse Kinematic Chain: Kinematics chain.1

Map Info

	Joints Map	Joints Type	Kin Axis Type	Kin DOF	Kin Part
Command 1 :	dof(1)	Rotational	Rot Z	1	R1_2
Command 2 :	dof(2)	Rotational	Rot Z	2	R2_1

Figure 26: *Configurations* and *Actuator Space Map* tabs

‘*Solver Attributes*’ tab is used in order to define the parameters for the chosen inverse kinematics method. In this tab, three sections are represented. First, the user is asked to define convergence tolerances for the numeric solution for both angular and linear convergence. In the second section, user decides on which joints to be solved, and in the last section, TCP convergence between robot TCP and target location is defined for X, Y, Z directions and Roll-Pitch-Yaw rotations. These sections are unique to each mechanism in hand. This tab can be seen in figure 27 for Exechon's RR wrist.


Execheon_RR_wrist's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

Convergence Tolerances
Linear : 0.100 mm
Angular : 0.010 deg

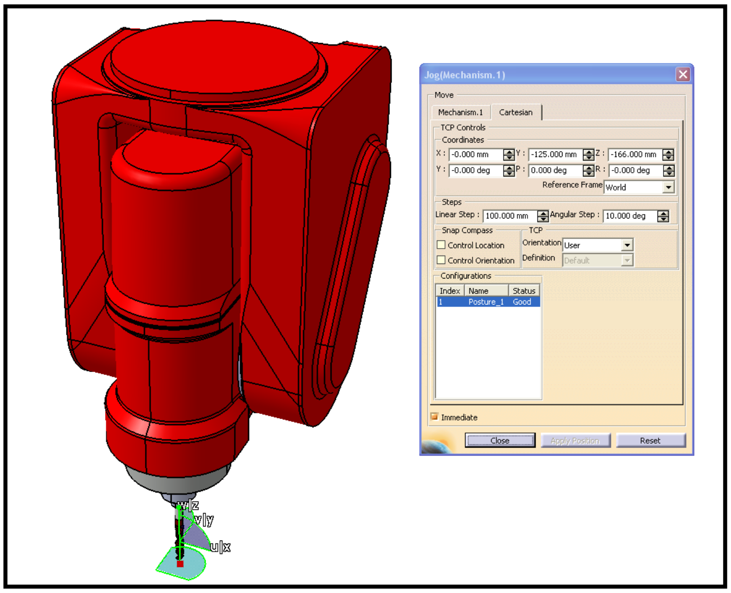
Solve Joints
Joint1: Yes
Joint2: Yes

TCP Convergence
X: Yes
Y: Yes
Z: Yes
Yaw: Yes
Pitch: Yes
Roll: Yes

Figure 27: *Solver Attributes* tab

After defining the necessary tabs, the inverse kinematics now is ready to use. To test its accuracy, click on '*Jog mechanism*' and the menu with a new tab called 'Cartesian' will prompt. The idea with this tab is to use a tag at the predefined TCP location to jog the robot to a certain point by using inverse kinematics calculation. In this tab, the user is allowed to change the TCP from defined point to any desired location as well as to jog the mechanism by using Cartesian coordinate system with respect to any defined coordinate frame. These features can be observed in figure 28.

This way a mechanism defined with inverse kinematics can be used in other modules of DELMIA for simulation purposes.



TCP Controls
Coordinates
X : -0.000 mm Y : -125.000 mm Z : -166.000 mm
Y : -0.000 deg P : 0.000 deg R : -0.000 deg
Reference Frame: World

Steps
Linear Step : 100.000 mm Angular Step : 10.000 deg

Snap Compass
☐ Control Location
☐ Control Orientation

TCP
Orientation: User
Definition: Default

Configurations

Index	Name	Status
1	Posture_1	Good

☑ Immediate

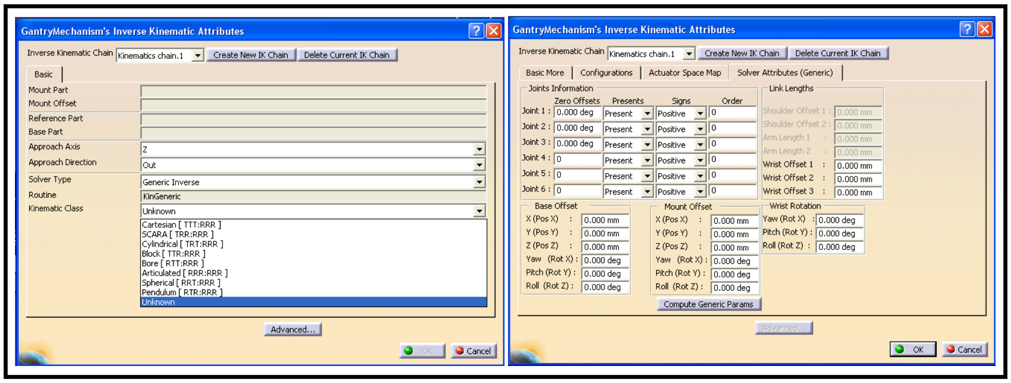
Figure 28: *Cartesian* tab with TCP tag

Generic method is used for most commonly adapted structures in the field robotics. These structures defined by DELMIA V5 as kinematic classes are

- Cartesian robot (TTT:RRR)
- SCARA robot (TRR:RRR)
- Cylindrical robot (TRT:RRR)
- Block robot (TTR:RRR)
- Bore robot (RTT:RRR)
- Articulated robot (RRR:RRR)
- Spherical robots (RRT:RRR)
- Pendulum robot (RTR:RRR)

The meaning of kinematic classes shown in parentheses is that the robot has 2 divided structures. The first structure is the body and shown before the colon. The part after the colon, on the other hand, represents the structure known as mount or wrist. For example, Cartesian robot has three translational joints in the body and this is represented as TTT. The RRR section, whereas, represents the three serially connected revolute joint as a wrist attached at the end-effector. The kinematic classes and the remaining properties for this method are shown in figure 29.

When ‘*Advanced’* is clicked the same tabs with numeric method appear, whereas the contents of the *Solver Attributes* tab are different. In this tab, four different sections are shown. First section is named ‘*Joints Information*’. In this section, the user defines *Offsets*, *Presents*, *Signs* and *Order*. The ‘Offsets’ are the distances of joints from the original coordinate frame of the joint. *Presents* are used to inform DELMIA V5 whether the joint should be included in inverse kinematics calculation. ‘*Signs’* section decide on the direction of translation or rotation whereas ‘*Order’* describes the calculation order that should be taken into account for the given kinematic class.



Inverse Kinematic Chain: Kinematics chain.1

Mount Part	
Mount Offset	
Reference Part	
Base Part	
Approach Axis	Z
Approach Direction	Out
Solver Type	Generic Inverse
Routine	KinGeneric
Kinematic Class	Unknown

Cartesian [TTT:RRR]
SCARA [TRR:RRR]
Cylindrical [TRT:RRR]
Block [TTR:RRR]
Bore [RTT:RRR]
Articulated [RRR:RRR]
Spherical [RRT:RRR]
Pendulum [RTR:RRR]
Unknown

Inverse Kinematic Chain: Kinematics chain.1

Joints Information

	Zero Offsets	Presents	Signs	Order
Joint 1 :	0.000 deg	Present	Positive	0
Joint 2 :	0.000 deg	Present	Positive	0
Joint 3 :	0.000 deg	Present	Positive	0
Joint 4 :	0	Present	Positive	0
Joint 5 :	0	Present	Positive	0
Joint 6 :	0	Present	Positive	0

Link Lengths

Shoulder Offset 1 :	0.000 mm
Shoulder Offset 2 :	0.000 mm
Arm Length 1 :	0.000 mm
Arm Length 2 :	0.000 mm
Wrist Offset 1 :	0.000 mm
Wrist Offset 2 :	0.000 mm
Wrist Offset 3 :	0.000 mm

Base Offset

X (Pos X) :	0.000 mm
Y (Pos Y) :	0.000 mm
Z (Pos Z) :	0.000 mm
Yaw (Rot X) :	0.000 deg
Pitch (Rot Y) :	0.000 deg
Roll (Rot Z) :	0.000 deg

Mount Offset

X (Pos X) :	0.000 mm
Y (Pos Y) :	0.000 mm
Z (Pos Z) :	0.000 mm
Yaw (Rot X) :	0.000 deg
Pitch (Rot Y) :	0.000 deg
Roll (Rot Z) :	0.000 deg

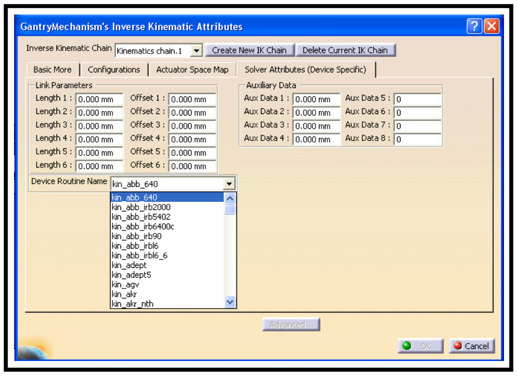
Wrist Rotation

Yaw (Rot X) :	0.000 deg
Pitch (Rot Y) :	0.000 deg
Roll (Rot Z) :	0.000 deg

Figure 29: Generic method and *Basic* and *Solver Attributes* tab

In the second section, ‘*Link Lengths*’, the offsets are used for rotational joints when their origins are coaxial. Shoulder offsets and arm lengths are only available for articulated robots. ‘*Base and Mount Offset*’ sections are used to define the transformation for any external coordinate frames that is set by the user for inverse kinematics calculation. ‘*Wrist Rotation*’ section describes the final rotation of TCP on the wrist. Unless changed, these values represent the same structure defined in forward kinematics building. When these values are set and clicked *OK*, the robot will be ready for simulation purposes.

Device-specific method is used for specific type of robots that are already defined in the library of DELMIA V5. Therefore, only difference of this method from *Generic Inverse* is that in *Solver Attributes* tab, DELMIA V5 asks its users to choose the routine name for the specific type of robot. The list of robots and their routines can be seen in figure 30.



Inverse Kinematic Chain: Kinematics chain.1

Link Parameters

Length 1 :	0.000 mm	Offset 1 :	0.000 mm
Length 2 :	0.000 mm	Offset 2 :	0.000 mm
Length 3 :	0.000 mm	Offset 3 :	0.000 mm
Length 4 :	0.000 mm	Offset 4 :	0.000 mm
Length 5 :	0.000 mm	Offset 5 :	0.000 mm
Length 6 :	0.000 mm	Offset 6 :	0.000 mm

Auxiliary Data

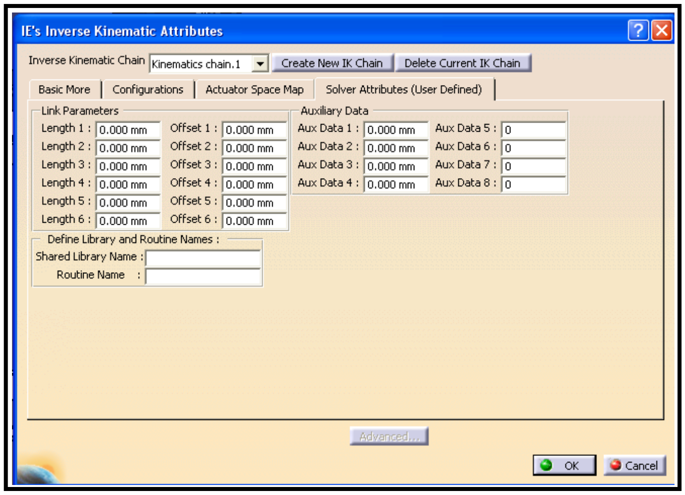
Aux Data 1 :	0.000 mm	Aux Data 5 :	0
Aux Data 2 :	0.000 mm	Aux Data 6 :	0
Aux Data 3 :	0.000 mm	Aux Data 7 :	0
Aux Data 4 :	0.000 mm	Aux Data 8 :	0

Device Routine Name: kin_abb_640

kin_abb_640
kin_abb_irb2000
kin_abb_irb5402
kin_abb_irb6400c
kin_abb_irb90
kin_abb_irbl6
kin_abb_irbl6_6
kin_adept
kin_adept5
kin_agv
kin_akr
kin_akr_nth

Figure 30: ***Device Specific*** **method and** ***Solver Attributes*** **tab**

The last and the topic of this paper is '*User-inverse*' method. This method is developed by DELMIA V5 in order to enable its users to integrate complicated calculations to a variety of mechanisms. The idea stems from the fact that some types of robots do not use widely known kinematic classes in their systems such as hybrid or parallel robots where inverse kinematics calculation cannot be solved by using regular approaches described in the beginning of this chapter. In order to select this method, as usual with other methods, solver type should be set to '*User inverse*'. When *Advanced* is clicked, the extra tabs *Configurations* and *Actuator Space Map* are the same as with other methods. On the other hand, *Solver Attributes* tab display differences. The differences can be seen in figure 31. The first section in this tab is '*Link Parameters*'. These parameters are used as input to C-code file to be utilized in the calculation. The 'Auxiliary Data' section also has the same properties as '*Link Parameters*'. The third section is '*Define Library and Routine Names*', which is where the user enters the name of the C-file as routine name and the library file created by compiling the code.



Inverse Kinematic Chain: Kinematics chain.1

Link Parameters

Length 1 :	0.000 mm	Offset 1 :	0.000 mm
Length 2 :	0.000 mm	Offset 2 :	0.000 mm
Length 3 :	0.000 mm	Offset 3 :	0.000 mm
Length 4 :	0.000 mm	Offset 4 :	0.000 mm
Length 5 :	0.000 mm	Offset 5 :	0.000 mm
Length 6 :	0.000 mm	Offset 6 :	0.000 mm

Auxiliary Data

Aux Data 1 :	0.000 mm	Aux Data 5 :	0
Aux Data 2 :	0.000 mm	Aux Data 6 :	0
Aux Data 3 :	0.000 mm	Aux Data 7 :	0
Aux Data 4 :	0.000 mm	Aux Data 8 :	0

Define Library and Routine Names :
Shared Library Name :
Routine Name :

Figure 31: ***User inverse*** **method and** ***Solver Attributes*** **tab**

In the next section, the structure of the C-file will be presented along with how to compile it for the use of DELMIA V5. Also this part will be covered in the appendix as how-to type documentation.

3.3.2. THE ANALYSIS OF C-FILE FOR INVERSE KINEMATICS IN DELMIA V5

In this section, the C-file structure will be analyzed. This analysis will not include any calculation or specific name but only kin_example. This name is chosen in order for a layman user to grasp the mechanics of C-file creation for DELMIA V5.

The C-code for inverse kinematics starts with a

```
#include <shlibdefs.h>
```

command. The *shlibdefs.h* file is a standard library for DELMIA V5 that has the standard macros and structures that are used in the creation of the inverse kinematics such as math operations.

```
#define NUM_SOLUTIONS    1         /* Number of possible solutions   */

#define NUM_DOFS         6         /* Number of joints to be solved */
```

These 2 lines of commands define the number of solutions achieved after solving the inverse kinematics (which also defines the number of possible postures a robot can perform for a given TCP) and degrees of freedom with command that the investigated mechanism has. The NUM_SOLUTIONS variable also defines the number of columns for the solution array, which will be shown later. The number of solutions and DOFs are determined by the robot type used and for example, a hexapod has six degrees of freedom with command and one possible solution for given TCP values.

In the following lines, the routines describe the interaction of the C-file with DELMIA V5. This interaction requires some change with respect to the name used for the C-file.

```
DllExport int
get_kin_config( char *kin_routine, int *kin_dof, int *solution_count, int
*usrKinDataHint )
{
    if( strcmp( kin_routine, "kin_example" ) == 0 )
        {
                *kin_dof = NUM_DOFS;
                *solution_count = NUM_SOLUTIONS;
                        /*
                        * this indicates kin_usr's last argument (void *pData)
                        * will be DLM_Data_KinStat
                        */
                        *usrKinDataHint = USR_KIN_DATA_KINSTAT;
                        return 0;
        }
        return 1;
}
```

```
static char JointType[2][24] = { "ROTATIONAL", "TRANSLATIONAL" };
static char KinMode[2][24] = { "Normal", "TrackTCP" };
```

In the following line, the user needs to state the name of C-file for *strcmp* command which compares the name of 2 strings and return 0 if the 2 strings match each other.

```
/*
 * User must supply this function
*/

DllExport int
get_kin_config( char *kin_routine, int *kin_dof, int *solution_count, int
*usrKinDataHint )
{
      if( strcmp( kin_routine, "kin_example" ) == 0 )
      {
            *kin_dof = NUM_DOFS;
            *solution_count = NUM_SOLUTIONS;
                  /*
                  * this indicates kin_usr's last argument (void *pData)
                  * will be DLM_Data_KinStat
                  */
                  *usrKinDataHint = USR_KIN_DATA_KINSTAT;
                  return 0;
      }
      return 1;
}
static char JointType[2][24] = { "ROTATIONAL", "TRANSLATIONAL" };
static char KinMode[2][24] = { "Normal", "TrackTCP" };
```

In this case, the name `"kin_example"` is the name of the C-file and it is stated in the *strcmp* command.

The following piece of code is utilized by DELMIA V5 to recognize the function named same as the C-file with variables which are input from DELMIA V5 to C-file. *'T6'* here is the transformation matrix. *'link_lengths'* is the distance between joint axes along the link lengths where "link offsets" is the shortest distance between joint axes – which are described in the previous section. These values according to DELMIA V5 are associated with the methodology called Denavitt-Hartenberg method (Hartenberg 1967).

```
/*
** Routine Name
*/
DllExport int
kin_hexapodFullTest(
      link_lengths,
      link_offsets,
      T6, /* See above for description of these arguments */
      solutions,
      warnings,
            pData
      )
/*
```

```
** Passed Variable Declarations
*/
double T6[4][4],
       link_lengths[],
       link_offsets[],
       solutions[][NUM_SOLUTIONS];
int warnings[];
```

For the lines above, a special attention should be given to the transformation matrix T6. As the name suggests, T6 is the result of successive multiplication of serial transformations, which are 3 translations in x, y and z direction, following 3 rotations about Z, Y and X-axes. Unlike, the traditional calculation of the transformation matrix for an articulated robot, T6 here is the direct transformation between the world coordinates attached or *Base Reference* depending on the choice of the user and the TCP.Thus, the transformation matrix as an input from DELMIA V5 has the form in figure 32.

$$\boxed{T6 = \begin{bmatrix} nx & ny & nz & 0 \\ ox & oy & oz & 0 \\ ax & ay & az & 0 \\ px & py & pz & 1 \end{bmatrix}}$$

Figure 32: T6 matrix of DELMIA V5

In this T6 matrix, the notations n(xyz), o(xyz) and a(xyz) represent the axes of the TCP. It must be noted that the representation of these axes have the row-vector form. Thus, when calculating the correct form of multiplication must be used. The p(xyz) notation describes the translation of a transformation matrix in X,Y and Z-axes with respect to the chosen coordinate frame as reference.

In the following piece of code, *pData* routine is defined. This routine is created as standard by DELMIA V5 in order for users to define their inverse kinematics; thus, *pData* routine is the main function for users. As seen below, the routine starts with the local variable declarations that are the constituting terms of the transformation matrix. The users are also entitled to add variables as they see fit for their calculation.

```
void *pData; /* usr routine should NEVER delete pData */

{
/*
** Local Variable Declarations (add variable declarations as appropriate)
*/

long double nx, ny, nz, ox, oy, oz, ax, ay, az, px, py, pz;
```

After variable declaration, DELMIA V5 inserts a standard if-loop to print mechanism properties and its current joint values for a given TCP. This part is essential for debugging purposes since these values are taken from '*Jog Mechanism*' window directly. This loop can be seen below.

```
#if 1
/*
* using pData
*/
      int i;

      DLM_Data_KinStat *pDLM_Data = (DLM_Data_KinStat *) pData;
      if( pDLM_Data )
      {
            printf( "\n\ndof_count: %d\n", pDLM_Data->dof_count );
            printf( "\njoint_types:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
            printf( "%s ", JointType[(pDLM_Data->joint_types)[i]] );
            printf( "\n\nkin_mode: %s\n", KinMode[pDLM_Data->kin_mode] );
            printf( "\njoint_values:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->joint_values[i] );
            printf( "\n\njnt_trvl_lmts lower:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[0][i] );
            printf( "\n\njnt_trvl_lmts upper:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[1][i] );
            printf( "\n\n" );
      }
#endif
```

The next section in the C-file is that DELMIA V5 declares that the users should start their calculation after this given point. The declaration is

```
/***--------------- Execution Begins Here ----------------------------------
***/
      /*
      ** DO NOT REMOVE THIS BLOCK OF CODE
      ** IT IS REQUIRED TO PROPERLY SET THE NUMBER OF KINEMATIC
      ** DOFS FOR THE DEVICE
      */
      if( !kin_check_definition( NUM_DOFS, NUM_SOLUTIONS ) )
      {
      /*
      ** Inconsistency between device definition and inverse
      ** kinematics routine exists. A warning message has been
      ** issued and routine aborted
      */
      return( 1 );
      }
/***---------------- User code begins here ---------------------------------
***/
```

After the necessary calculations are made, the user needs to feed DELMIA V5 back with the joint values. This is achieved by using an array called `solutions[][NUM_SOLUTIONS]`. An example of such action is given below.

```
        solutions[0][0] = J1;
        solutions[1][0] = J2;
        solutions[2][0] = J3;
        solutions[3][0] = J4;
        solutions[4][0] = J5;
        solutions[5][0] = J6;
```

The lines above appoint values to the elements of an array, where these values are named as J1, J2, etc. These elements belong to the current values of joints, in this case the six joints of a respective mechanism.

Next important aspect is that the users are also entitled to print any value on debugging window. This action can be delivered with a line, for example

```
printf( "J1 J2 J3: %12.4f ,%12.4f ,%12.4f\n", J1 ,J2 ,J3 );
printf( "J4 J5 J6: %12.4f ,%12.4f ,%12.4f\n", J4 ,J5 ,J6 );
```

After having finished calculations, the user also must supply the following line to inform DELMIA V5 that the results end in an accepted posture for the mechanism. This is done by feeding back the warnings array.

```
        warnings[ 0 ] = WARN_GOOD_SOLUTION;
        return (0);

} /* End of kin_example */
```

With the above lines, the user ends the creation of C-file and proceeds to compile the file for the creation of the required library files.

3.3.3. COMPILATION OF C-FILES

The compilation process of C-files is somewhat delicate; but once it is completed, the process itself becomes easy to repeat. Before, going deep it must be noted that a compilation tool is necessary for this operation. In this thesis work, Microsoft Visual Studio 8 is used as compilation tool. The compatibility of other tools has not been tested. Thus, the approved and recommended tool is Microsoft's Visual Studio (version of this program should no longer be earlier than VS 8). If another compiler has been chosen, it is important that the compilation tool must support C# language and has *nmake* feature available as a compilation operation is done via '*nmake all*' command.

Another important point regarding compilation process is about the operating system (OS) of the computer on which the simulation is going to be executed. If the system is 64-bit, users should implement a prerequisite operation before compilation. This operation will be covered in the appendix section as environment set-up. The reason to include this step in the appendix (APPENDIX F: COMPILATION OF C-FILES) is that it would be easier for layman users to follow a how-to type document.

4. THEORY OF INVERSE KINEMATICS FOR RESPECTIVE ROBOTS

In this chapter, the inverse kinematics for hexapod/Flexapod 6P, Exechon and Gantry-Tau robots will be given. Also, the transfer of the theory to C-file will be presented at the end of the theory for each robot.

4.1. HEXAPOD/FLEXAPOD 6P INVERSE KINEMATICS

The idea behind the inverse kinematics for hexapods is somewhat simple. The method relies on vector summation and with a known TCP position and orientation the leg lengths can be calculated as vectors and normalized to reach the total length.

As presented earlier, a hexapod system has the kinematics chain of RR:P:RRR for one leg which specifically stands for revolute-revolute-prismatic-revolute-revolute-revolute (Ji 2001). This chain is illustrated in figure 33.

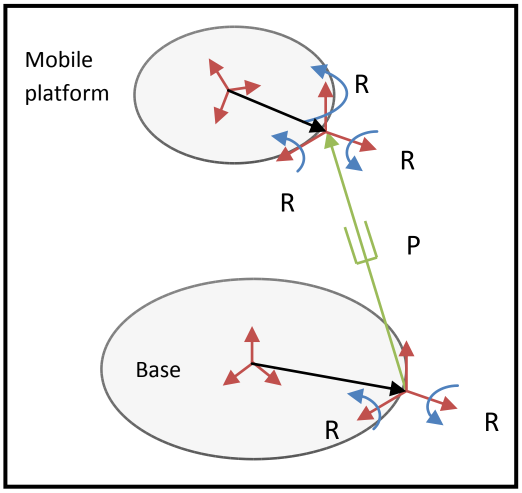


Figure 33: Hexapod/Flexapod kinematic chain of one leg

In any theoretical representation, it is important to first clarify the notation used for the inverse calculation. To start with, two different coordinate frames will be appointed. The first one XYZ_0 will be attached to the base platform that will be the fixed part of hexapod. The second frame XYZ_6 will be at the center of mobile platform. The first set of vectors $L_i b$ will be utilized to describe the position of attachment points of legs from the base platform. The second set of vectors L_i will describe the legs and the last set of vectors $L_i tToTCP$ will illustrate legs' upper attachment points with respect to the mobile platform's coordinate frame XYZ_6. The last vector is the position vector of the mobile platform notated as P_{XYZ}. These vectors are shown in figure 34. With the vectors at hand, the following summation can be formulated (Yang 1998).

$$\overrightarrow{L_\iota} = R\ x\ \overrightarrow{L_\iota tToTCP} + \overrightarrow{P_{XYZ}} - \overrightarrow{L_\iota b} \qquad (3.1)$$

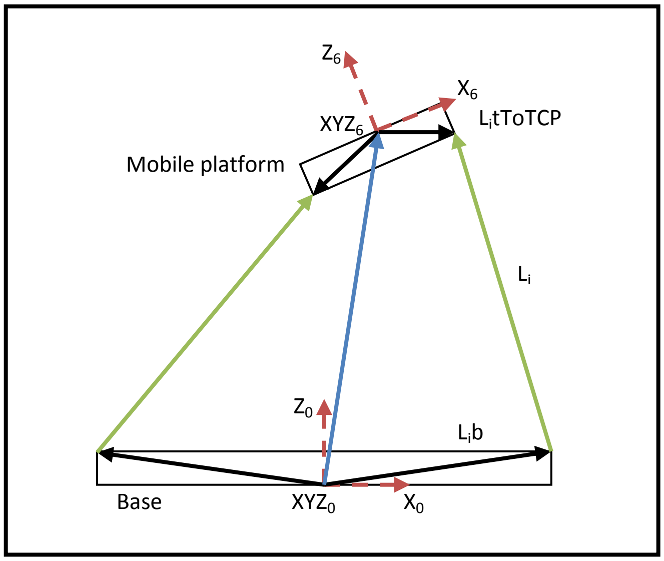


Figure 34: Hexapod and the constituting vectors

What equation 3.1 aims is that it describes the L1tToTCP vector in the base coordinates XYZ_0 by multiplying it with the rotation matrix. Then by adding the translation vector P_{XYZ}, It reaches to the upper attachment point. By subtracting L_1b from the summation, the result becomes the vector between lower attachment point and the upper one, which is the vector L_i. (Yang 1998).

Then, to reach the total length of the leg normalization of the vector should be done by

$$|\overrightarrow{L_i}| = \sqrt{{L_{i_x}}^2 + {L_{i_y}}^2 + {L_{i_z}}^2} \tag{3.2}$$

Before transferring the theory to a C-file, it is important to give the coordinates of the vectors defined earlier. These vectors are constant and L_ib is defined with respect to XYZ_0 whereas $L_itToTCP$ is defined with respect to XYZ_6. These vectors and their coordinates are given in table 1 and table 2for Hexapod and Flexapod 6P.

HEXAPOD	Lower Attachment Points $L_i b$						Upper Attachment Points $L_i tToTCP$					
Coordinates	**Leg 1**	**Leg 2**	**Leg 3**	**Leg 4**	**Leg 5**	**Leg 6**	**Leg 1**	**Leg 2**	**Leg 3**	**Leg 4**	**Leg 5**	**Leg 6**
X-coordinate	31	-31	-117.826	-86.826	86.826	117.826	31	- 31	-57.761	-26.761	26.761	57.761
Y-coordinate	118.156	118.156	-32.231	-85.925	-85.925	-32.231	48.799	48.799	2.447	-51.246	-51.246	2.447
Z-coordinate	40.205	40.205	40.205	40.205	40.205	40.205	-31.45	-31.45	-31.45	-31.45	-31.45	-31.45

Table 1: Coordinates of vectors for hexapod

FLEXAPOD 6P	Lower Attachment Points $L_i b$						Upper Attachment Points $L_i tToTCP$					
Coordinates	**Leg 1**	**Leg 2**	**Leg 3**	**Leg 4**	**Leg 5**	**Leg 6**	**Leg 1**	**Leg 2**	**Leg 3**	**Leg 4**	**Leg 5**	**Leg 6**
X-coordinate	-132.5	43.733	88.767	88.767	43.733	-132.5	-48.767	-3.733	52.5	52.5	-3.733	-48.767
Y-coordinate	26	127.748	101.748	-101.748	-127.748	-26	32.466	58.466	26	-26	-58.466	-32.466
Z-coordinate	58.5	58.5	58.5	58.5	58.5	58.5	-75	-75	-75	-75	-75	-75

Table 2: Coordinates of vectors for Flexapod 6P

4.2. HEXAPOD/FLEXAPOD 6P – TRANSFER OF THEORY TO A C-FILE

In this section, the theory presented in 4.1 will be transferred to the C-file. Only the theory and calculations will be described and complete C-file can be seen in appendix.

The calculation starts with, as stated in 3.3.2, the necessary declarations for variables. These variables are the elements of the transformation matrix, leg lengths and legs upper attachment coordinates. The corresponding notation, then,

- Transformation matrix: nx, ny, nz, ox, oy, oz, ax, ay, az, px, py, pz
- Total leg lengths: L1, L2, L3, L4, L5, L6
- Coordinates:
    - Leg 1: D11 in X-axis, D12 in Y-axis, D13 in Z-axis
    - Leg 2: D21 in X-axis, D22 in Y-axis, D23 in Z-axis
    - Leg 3: D31 in X-axis, D32 in Y-axis, D33 in Z-axis
    - Leg 4: D41 in X-axis, D42 in Y-axis, D43 in Z-axis
    - Leg 5: D51 in X-axis, D52 in Y-axis, D53 in Z-axis
    - Leg 6: D61 in X-axis, D62 in Y-axis, D63 in Z-axis
- Joint values: J1, J2, J3, J4, J5, J6
- Leg length when joint command is zero: Lref
- Vectors:
    - Vectors that connect mobile platform’s coordinate frame to upper attachment points
        - L1tToTCP[4][1]
        - L2tToTCP[4][1]
        - L3tToTCP[4][1]
        - L4tToTCP[4][1]
        - L5tToTCP[4][1]
        - L6tToTCP[4][1]
    - Vectors that connect base platform to lower attachment points
        - L1b[4][1]
        - L2b[4][1]
        - L3b[4][1]
        - L4b[4][1]
        - L5b[4][1]
        - L6b[4][1]
    - Intermediary vector definition that is the result of the summation between $\overrightarrow{L_{\imath}tCur} = R\ x\ \overrightarrow{L_{\imath}tToTCP} + \overrightarrow{P_{XYZ}}$
        - L1tCur[4]
        - L2tCur[4]
        - L3tCur[4]
        - L4tCur[4]
        - L5tCur[4]

        - L6tCur[4]
    - Array declaration for the arranged transformation matrix which is in column vector form
        - TCP[4][4]
    - Additional variables to perform matrix multiplication
        - inner1, inner2, inner3, inner4, inner5, inner6
        - row1, row2, row3, row4, row5, row6
        - col1, col2, col3, col4, col5, col6

These declarations in the code should be as

```
void *pData; /* usr routine should NEVER delete pData */

{
/*
** Local Variable Declarations (add variable declarations as appropriate)
*/
long double nx, ny, nz, ox, oy, oz, ax, ay, az, px, py, pz;
long double D11, D12, D13, D21, D22, D23, D31, D32, D33, D41, D42, D43, D51,
D52, D53, D61, D62, D63;
long double L1,L2,L3,L4,L5,L6,J1,J2,J3,J4,J5,J6, Lref;

//Variables to perform matrix multiplication
int row1,row2,row3,row4,row5,row6;
int col1,col2,col3,col4,col5,col6;
int inner1,inner2,inner3,inner4,inner5,inner6;

// The upper attachmentpoints for each leg (The vector between the TCP and
each upper attachment point).
long double L1tToTCP[4][1];
long double L2tToTCP[4][1];
long double L3tToTCP[4][1];
long double L4tToTCP[4][1];
long double L5tToTCP[4][1];
long double L6tToTCP[4][1];

//The transformed T6 matrix named as TCP matrix (do not confuse with Tool
Centre Point) - see line 307.
long double TCP[4][4];

//The current position for each upper attachmentpoint (in Base coordinates).
long double L1tCur[4] = {0};
long double L2tCur[4] = {0};
long double L3tCur[4] = {0};
long double L4tCur[4] = {0};
long double L5tCur[4] = {0};
long double L6tCur[4] = {0};

//Lower attachemnt points on each leg (in Base coordinates)
long double L1b[3];
long double L2b[3];
long double L3b[3];
long double L4b[3];
long double L5b[3];
long double L6b[3];
```

After the declarations, matrix elements should be appointed to T6 matrix – which is, as said earlier, standard definition and input of DELMIA V5 to describe the transformation between two coordinate frames. This operation is done by

```
//Importing the current TCP values from Delmia through the T6 matrix and
putting proper context
nx = T6[0][0];
ny = T6[0][1];
nz = T6[0][2];
ox = T6[1][0];
oy = T6[1][1];
oz = T6[1][2];
ax = T6[2][0];
ay = T6[2][1];
az = T6[2][2];
px = T6[3][0];
py = T6[3][1];
pz = T6[3][2];
```

In order to perform the matrix multiplication, the transformation matrix should be rearranged in column vector format. This operation can be skipped and the rest of the calculation can be done accordingly with the row vector form; however, for this thesis work column vector form is chosen. So this arrangement is done via

```
//The transforming T6 matrix from row vectors form to column vector form
      TCP[0][0] =  nx; TCP[0][1] =  ox; TCP[0][2] =  ax; TCP[0][3] = px;
      TCP[1][0] =  ny; TCP[1][1] =  oy; TCP[1][2] =  ay; TCP[1][3] = py;
      TCP[2][0] =  nz; TCP[2][1] =  oz; TCP[2][2] =  az; TCP[2][3] = pz;
      TCP[3][0] =   0;  TCP[3][1] = 0;   TCP[3][2] =  0;  TCP[3][3] = 1;
```

At this point, by measuring the coordinates of upper and lower attachment points in hexapod when all the actuators are zero, L_itToTCP and L_ib vectors can be defined with actual vector values. Thus the corresponding values are declared as

```
//The vector between the TCP and the upper attachmentpoints for each leg
L1tToTCP[0][0] =      31;  L1tToTCP[1][0] =  48.799; L1tToTCP[2][0] = -31.45;
L1tToTCP[3][0] =       1;
L2tToTCP[0][0] =     -31; L2tToTCP[1][0] =   48.799; L2tToTCP[2][0] = -31.45;
L2tToTCP[3][0] =       1;
L3tToTCP[0][0] = -57.761; L3tToTCP[1][0] =    2.447; L3tToTCP[2][0] = -31.45;
L3tToTCP[3][0] =       1;
L4tToTCP[0][0] = -26.761; L4tToTCP[1][0] =  -51.246; L4tToTCP[2][0] = -31.45;
L4tToTCP[3][0] =       1;
L5tToTCP[0][0] =  26.761; L5tToTCP[1][0] =  -51.246; L5tToTCP[2][0] = -31.45;
L5tToTCP[3][0] =       1;
L6tToTCP[0][0] =  57.761; L6tToTCP[1][0] =    2.447; L6tToTCP[2][0] = -31.45;
L6tToTCP[3][0] =       1;
//The lower attachmentpoints for each leg (in Base-coordinates).
L1b[0]      = 31;       L1b[1] = 118.156;       L1b[2] = 40.205;
L2b[0]      =-31;       L2b[1] = 118.156;       L2b[2] = 40.205;
L3b[0]      =-117.826;  L3b[1] = -32.231;       L3b[2] = 40.205;
L4b[0]      =-86.826;   L4b[1] = -85.925;       L4b[2] = 40.205;
L5b[0]      = 86.826;   L5b[1] = -85.925;       L5b[2] = 40.205;
```

```
L6b[0]      = 117.826;  L6b[1] = -32.231;       L6b[2] = 40.205;
Lref = 376.5;
```

Since the coordinates are appointed to the vectors, equation 3.1 can be executed. Thus the multiplication and summation $R\ x\ \overrightarrow{L_\iota tToTCP} + \overrightarrow{P_{XYZ}}$ is done with for six legs where R matrix is TCP

```
//Calculating the current position (in x,y,z in Base coordinates) of each
upper attachment point for each leg by multiplying the transformation
// matrix TCP[4][4] with the vector between the current TCP (the T6 matrix)
and the upper attachmentpoint for each leg (LxToTCP[][])

//Calculate upper position on Leg1 (The array L1tCur)
 for (row1 = 0; row1 < 4; row1++) {
        for (col1 = 0; col1 < 1; col1++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner1 = 0; inner1 < 4; inner1++) {
                L1tCur[row1] += TCP[row1][inner1] * L1tToTCP[inner1][col1];
            }
        }
  }
 //Calculate upper position on Leg2 (The array L2tCur)
    for (row2 = 0; row2 < 4; row2++) {
        for (col2 = 0; col2 < 1; col2++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner2 = 0; inner2 < 4; inner2++) {
                L2tCur[row2] += TCP[row2][inner2] * L2tToTCP[inner2][col2];
            }
        }
  }
//Calculate upper position on Leg3 (The array L3tCur)
    for (row3 = 0; row3 < 4; row3++) {
        for (col3 = 0; col3 < 1; col3++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner3 = 0; inner3 < 4; inner3++) {
                L3tCur[row3] += TCP[row3][inner3] * L3tToTCP[inner3][col3];
            }
        }
  }
//Calculate upper position on Leg4 (The array L4tCur)
    for (row4 = 0; row4 < 4; row4++) {
        for (col4 = 0; col4 < 1; col4++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner4 = 0; inner4 < 4; inner4++) {
                L4tCur[row4] += TCP[row4][inner4] * L4tToTCP[inner4][col4];
            }
        }
  }
```

```
//Calculate upper position on Leg5 (The array L5tCur)
    for (row5 = 0; row5 < 4; row5++) {
        for (col5 = 0; col5 < 1; col5++) {
            // Multiply the row of A by the column of B to get the row, 
column of product.
            for (inner5 = 0; inner5 < 4; inner5++) {
                L5tCur[row5] += TCP[row5][inner5] * L5tToTCP[inner5][col5];
            }
        }
  }
//Calculate upper position on Leg6 (The array L6tCur)
    for (row6 = 0; row6 < 4; row6++) {
        for (col6 = 0; col6 < 1; col6++) {
            // Multiply the row of A by the column of B to get the row, 
column of product.
            for (inner6 = 0; inner6 < 4; inner6++) {
                L6tCur[row6] += TCP[row6][inner6] * L6tToTCP[inner6][col6];
            }
        }
  }
```

The result of this operation is L_itCur. Since the transformation matrix is 4x4 in which translation vector $\overrightarrow{P_{XYZ}}$ is included, the summation operation is automatically done as the multiplication operation continues. So L_itCur is then

$$L_i tCur = R\ x\ \overrightarrow{L_\imath tToTCP} + \overrightarrow{P_{XYZ}} = \ TCPx\ \overrightarrow{L_\imath tToTCP} \tag{3.3}$$

Then, the subtraction operation will be done. With the result of the subtraction operation in hand, it is instantly normalized by adding the squares of vector components of the resulting vector. This is achieved via

```
// Calcultates the distance between the upper and lower attachment points for 
each leg.
L1 = sqrt(((pow((L1tCur[0]-L1b[0]),2)))+((pow((L1tCur[1]-
L1b[1]),2)))+((pow((L1tCur[2]-L1b[2]),2))));

L2 = sqrt(((pow((L2tCur[0]-L2b[0]),2)))+((pow((L2tCur[1]-
L2b[1]),2)))+((pow((L2tCur[2]-L2b[2]),2))));

L3 = sqrt(((pow((L3tCur[0]-L3b[0]),2)))+((pow((L3tCur[1]-
L3b[1]),2)))+((pow((L3tCur[2]-L3b[2]),2))));

L4 = sqrt(((pow((L4tCur[0]-L4b[0]),2)))+((pow((L4tCur[1]-
L4b[1]),2)))+((pow((L4tCur[2]-L4b[2]),2))));

L5 = sqrt(((pow((L5tCur[0]-L5b[0]),2)))+((pow((L5tCur[1]-
L5b[1]),2)))+((pow((L5tCur[2]-L5b[2]),2))));

L6 = sqrt(((pow((L6tCur[0]-L6b[0]),2)))+((pow((L6tCur[1]-
L6b[1]),2)))+((pow((L6tCur[2]-L6b[2]),2))));
```

Then from total leg lengths, the reference length (notated as Lref) – which is the total length when joint command is zero – will be subtracted. This way, joint values will be achieved and these values J1, J2, J3, J4, J5 and J6 will be fed back to '*solutions*' array. This is accomplished by

```
//Calculates the joint values by calulating the differnce in distance between
the two attachmentpoints on each leg and a reference length (Lref)
//(the lenght between attachment points when the joints are 0)
J1 =  L1 - Lref;
J2 =  L2 - Lref;
J3 =  L3 - Lref;
J4 =  L4 - Lref;
J5 =  L5 - Lref;
J6 =  L6 - Lref;

D11 = L1tCur[0]; D12 = L1tCur[1]; D13 = L1tCur[2];
D21 = L2tCur[0]; D22 = L2tCur[1]; D23 = L2tCur[2];
D31 = L3tCur[0]; D32 = L3tCur[1]; D33 = L3tCur[2];
D41 = L4tCur[0]; D42 = L4tCur[1]; D43 = L4tCur[2];
D51 = L5tCur[0]; D52 = L5tCur[1]; D53 = L5tCur[2];
D61 = L6tCur[0]; D62 = L6tCur[1]; D63 = L6tCur[2];

//Sending the final joint values back to the "solutions"-matrix which is the
input matrix for Delmia.
solutions[0][0] = J1;
solutions[1][0] = J2;
solutions[2][0] = J3;
solutions[3][0] = J4;
solutions[4][0] = J5;
solutions[5][0] = J6;
```

With the lines above, the calculation phase is accomplished. After this point, the user can also print any value on debugging window in order to verify that the inverse kinematics is working. Such printing operation, in this case, can be done for leg lengths and their corresponding coordinates with

```
//Printing some of the variable values out in the debug window to ease
debugging and get an overview of what is going on
printf( "\n The leg lengths\n" );
printf( "J1 J2 J3: %12.4f ,%12.4f ,%12.4f\n", J1 ,J2 ,J3 );
printf( "J4 J5 J6: %12.4f ,%12.4f ,%12.4f\n", J4 ,J5 ,J6 );
printf( "L1 L2 L3: %12.4f ,%12.4f ,%12.4f\n", L1 ,L2 ,L3 );
printf( "L4 L5 L6: %12.4f ,%12.4f ,%12.4f\n", L4 ,L5 ,L6);
printf( "\n The legs' upper attachment point coordinates \n" );
printf( "\D11 D12 D13: %12.4f ,%12.4f ,%12.4f\n", D11 ,D12 ,D13 );
printf( "\D21 D22 D23: %12.4f ,%12.4f ,%12.4f\n", D21 ,D22 ,D23 );
printf( "\D31 D32 D33: %12.4f ,%12.4f ,%12.4f\n", D31 ,D32 ,D33 );
printf( "\D41 D42 D43: %12.4f ,%12.4f ,%12.4f\n", D41 ,D42 ,D43 );
printf( "\D51 D52 D53: %12.4f ,%12.4f ,%12.4f\n", D51 ,D52 ,D53 );
printf( "\D61 D62 D63: %12.4f ,%12.4f ,%12.4f\n", D61 ,D62 ,D63 );
```

After printing values the code continues with the declaration of an acceptable solution via

```
warnings[ 0 ] = WARN_GOOD_SOLUTION;
```

And the C-file is terminated with

```
return (0);
}
```

With termination, the transfer of the theory is completed. It must be noted that the coordinates used in this code belong to hexapod. For Flexapod 6P case, the corresponding lines can be altered according to values in Table 2.

4.3. EXECHON INVERSE KINEMATICS – THEORY

The inverse kinematics theory of Exechon has been presented in literature in various ways. One of those ways is presented by Bi (2011). In the respective theory, inverse kinematics is treated by using intermediate variables defining the position and orientation of the mobile platform; and through these variables, the coordinates of the attachment points are calculated. Zoppi (2010) is also adopting a similar way where the inverse kinematics approach is defined via intermediate variables that are not defining the direct value of the joints. Thus, a parallel theory employed for Stewart platforms can be applied to Exechon.

The idea of this similar inverse kinematics is that with a transformation matrix, the vectors that connect mobile platform to legs' lower attachment points can be described with respect to base coordinate frame, which is done by multiplication of these vectors with a T6 matrix. Then, the constant vectors that connect the base coordinate frame to fixed upper attachment points – which are the centers of RR-joints for identical legs and spherical joint for the perpendicular leg – are subtracted from the product of the multiplication operation. Then, the resulting vectors that define the legs are normalized and the standard leg lengths when the actuators are zero are subtracted from the normalized vectors – which results in the joint values under actuation (Bi 2011).

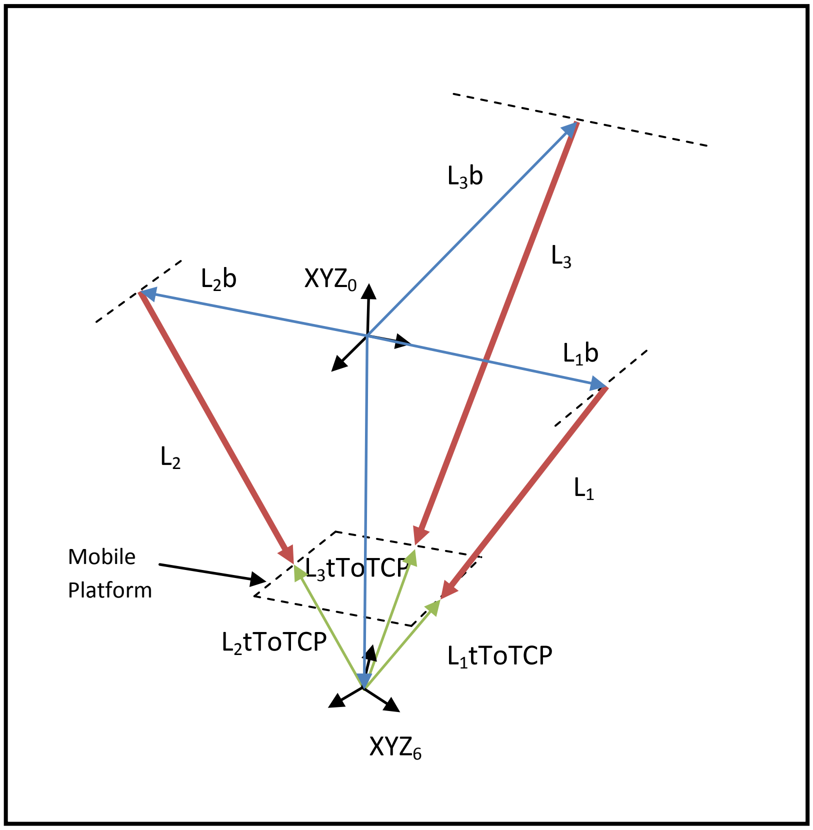


Figure 35: The vector description of Exechon robot's parallel structure

In order to clarify the calculations, the vectors need to be illustrated with their notations. These vectors are shown in figure 35. Then, the theory of inverse kinematics for the parallel part of Exechon will be presented.

The necessary vectors and the coordinate frames are then

- XYZ_0 – Base coordinate system
- XYZ_6 – Mobile platform's coordinate system located at the center of a possible wrist
- L_i – The set of vectors to define the corresponding leg's position and orientation.
- L_ib – The constant vectors that define the location of upper attachment points of legs
- L_itToTCP – The set of vectors that connect XYZ_6 to attachment points on mobile platform

As put earlier, the methodology is somewhat similar to hexapod; and that is the reason same notation is used for this robot as well.

First step in the calculation is to describe L_itToTCP vectors in the base coordinate system (Bi 2011). To achieve that, the use of transformation matrix T6 – which is input from DELMIA V5 – is required. The transformation matrix for this robot will be measuring the coordinate frame XYZ_6's orientation and position. Thus, description of the mobile platform's vectors in base coordinate frame is

$$\overrightarrow{{}^{0}(L_\iota\mathrm{tToTCP})} = {}^{0}_{6}\mathrm{T6} \times \overrightarrow{{}^{6}(L_\iota\mathrm{tToTCP})} \tag{3.4}$$

Then, since transformation matrix includes translation the result of equation 3.4 is now equal to the summation of L_ib and L_i. When L_ib is subtracted, the result will be the vector describing L_i (Bi 2011). Hence,

$$\overrightarrow{{}^{0}(L_\iota\mathrm{tToTCP})} = \overrightarrow{L_\iota b} + \overrightarrow{L_\iota} \tag{3.5}$$

$$\overrightarrow{L_\iota} = \overrightarrow{{}^{0}(L_\iota\mathrm{tToTCP})} - \overrightarrow{L_\iota b} \tag{3.6}$$

Finally, the normalization of this vector will yield the total length, which is

$$\left|\overrightarrow{L_\iota}\right| = \sqrt{{L_{i_x}}^2 + {L_{i_y}}^2 + {L_{i_z}}^2} \tag{3.7}$$

In the case of Exechon as with hexapod, the reference length must be subtracted from the product of equation 3.7 so that the result of subtraction can be fed back to solutions matrix of the C-code. This part is not described in the theory section for it is defined when the assembly is completed in DELMIA V5. However, it must be noted that the reference lengths of Exechon robot may differ from each other depending on the CAD-models. These lengths in this thesis work are 803.887 mm for identical legs and 886.021 mm for leg 3. As in the case of hexapod, these constant vectors should be defined and these values for the coordinates of the respective vectors are given in table 3.

HEXAPOD	From Base Coordinate System to Upper Attachment Points L_ib			Mobile Platform's Lower Attachment Points L_itToTCP		
Coordinates	**Leg 1**	**Leg 2**	**Leg 3**	**Leg 1**	**Leg 2**	**Leg 3**
X-coordinate	420	-420	0	173	-173	0
Y-coordinate	0	0	670	-50	-50	173
Z-coordinate	0	0	0	485	485	485

Table 3: Coordinates for the vectors of Exechon

4.4. THEORY OF EXECHON TO C-FILE

As with the case of Flexapod 6P and hexapod cases, the c-file starts with declarations of coordinates. These coordinates and corresponding vectors are declared as

```
//The vector between the TCP and the upper attachmentpoints for each leg
L1tToTCP[0][0] =    173; L1tToTCP[1][0] =    -50; L1tToTCP[2][0] = 485;
L1tToTCP[3][0] = 1;

L2tToTCP[0][0] =   -173; L2tToTCP[1][0] =    -50; L2tToTCP[2][0] = 485;
L2tToTCP[3][0] = 1;

L3tToTCP[0][0] =      0; L3tToTCP[1][0] =    173; L3tToTCP[2][0] = 485;
L3tToTCP[3][0] = 1;

//The lower attachmentpoints for each leg (in Base-coordinates).
L1b[0]      =     420;      L1b[1] =   0;         L1b[2] = 0;
L2b[0]      =    -420;      L2b[1] =   0;         L2b[2] = 0;
L3b[0]      =       0;      L3b[1] = 670;         L3b[2] = 0;
```

Then, the calculations start with transforming or in other words describing L_itToTCP vectors in base coordinate system by using a matrix multiplication function for three L_itToTCP vectors.

```
//Calculate upper position on Leg1 (The array L1tCur)
 for (row1 = 0; row1 < 4; row1++) {
        for (col1 = 0; col1 < 1; col1++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner1 = 0; inner1 < 4; inner1++) {
                L1tCur[row1] += TCP[row1][inner1] * L1tToTCP[inner1][col1];
            }
        }
  }
 //Calculate upper position on Leg2 (The array L2tCur)
    for (row2 = 0; row2 < 4; row2++) {
        for (col2 = 0; col2 < 1; col2++) {
            // Multiply the row of A by the column of B to get the row, 
column of product.
            for (inner2 = 0; inner2 < 4; inner2++) {
                L2tCur[row2] += TCP[row2][inner2] * L2tToTCP[inner2][col2];
            }
        }
  }
//Calculate upper position on Leg3 (The array L3tCur)
    for (row3 = 0; row3 < 4; row3++) {
        for (col3 = 0; col3 < 1; col3++) {
            // Multiply the row of A by the column of B to get the row, 
column of product.
            for (inner3 = 0; inner3 < 4; inner3++) {
                L3tCur[row3] += TCP[row3][inner3] * L3tToTCP[inner3][col3];
            }
        }
  }
```

Then, from L_itToTCP vectors the base vectors L_ib will be subtracted and the product of this operation will be normalized. This operation is done via

```
// Calcultates the distance between the upper and lower attachment points for 
each leg.
L1 = sqrt(((pow((L1tCur[0]-L1b[0]),2)))+((pow((L1tCur[1]-
L1b[1]),2)))+((pow((L1tCur[2]-L1b[2]),2))));

L2 = sqrt(((pow((L2tCur[0]-L2b[0]),2)))+((pow((L2tCur[1]-
L2b[1]),2)))+((pow((L2tCur[2]-L2b[2]),2))));

L3 = sqrt(((pow((L3tCur[0]-L3b[0]),2)))+((pow((L3tCur[1]-
L3b[1]),2)))+((pow((L3tCur[2]-L3b[2]),2))));
```

From this total length, the reference length of for identical legs and leg 3 will be subtracted and the result will be fed back to solutions matrix.

```
//The distance between upper and lower leg attachmentpoint when the command 
joint is zero. Used as a reference to get the current leg length
Lref12 = 803.887;
Lref3 = 886.021;
//Calculates the joint values by calulating the differnce in distance between 
the two attachmentpoints on each leg and a reference length (Lref)
//(the lenght between attachment points when the joints are 0)
J1 =  L1 - Lref12;
J2 =  L2 - Lref12;
J3 =  L3 - Lref3;
//Sending the final joint values back to the "solutions"-matrix which is the 
input matrix for Delmia.
solutions[0][0] = J1;
solutions[1][0] = J2;
solutions[2][0] = J3;
```

4.5. GANTRY-TAU ROBOT

The inverse kinematics aspect of Gantry-Tau robots bears some similarities to previous robots of this thesis work since it is comprised of a parallel structure. The schematic description of this robot can be seen in figure 36.

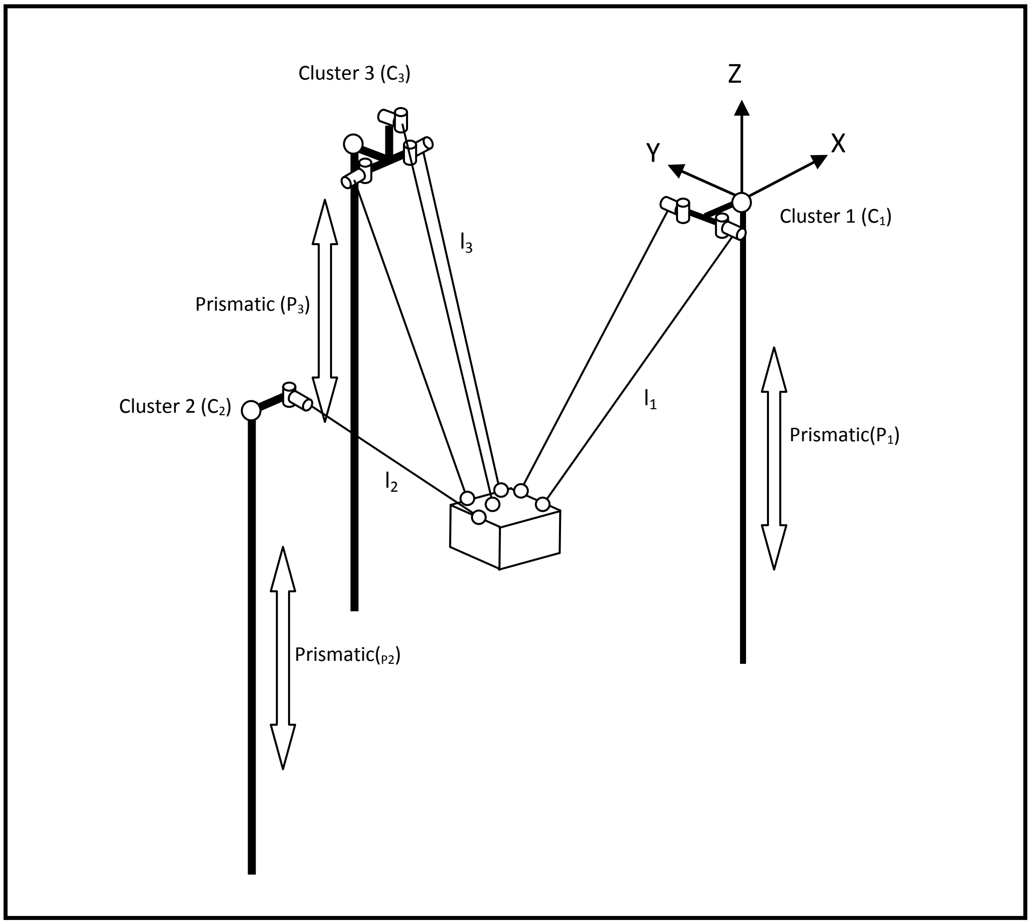


Figure 36: Gantry-Tau robot

In order to develop the inverse kinematics, the kinematic description of Gantry-Tau is essential. Each link in the clusters have the kinematic chain of prismatic joint with actuation, universal joint and a spherical joint that connects the link to the mobile platform. Therefore, from the inverse kinematics perspective, the outcome of calculations should yield values for prismatic actuators P_i (i = 1-3). To find these joint values, then, each cluster will be analyzed separately; and in each cluster, one link and its constituting vectors will be used due to parallel formation of the mechanism (Johannesson 2003).

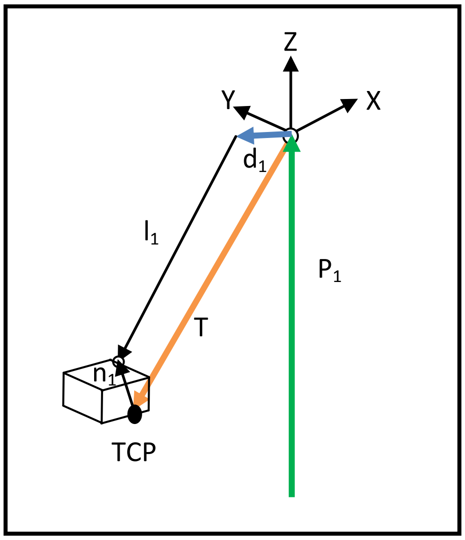


Figure 37: Link 1's vector definition

In figure 37, the vectors that constitute link 1's kinematic chain can be seen. From these vectors, it is possible to develop the equation 3.8 for link 1

$$\overrightarrow{P_1} + \overrightarrow{d_1} + \overrightarrow{l_1} = {}_6^0\mathrm{T6} \times \overrightarrow{\mathrm{n}_1} \tag{3.8}$$

$${}_6^0\mathrm{T6} \times \overrightarrow{\mathrm{n}_1} = \vec{\mathrm{T}} + {}_6^0\mathrm{R6} \times \overrightarrow{\mathrm{n}_1} \tag{3.9}$$

where ${}_6^0\mathrm{T6}$ is the transformation matrix between the TCP and base frame whereas ${}_6^0\mathrm{R6}$ is the rotational part of ${}_6^0\mathrm{T6}$. Since l_1 is a constant length in the mechanism as well as n_1 and d_1, the actuators coordinate's coordinates can easily be found (Johannesson 2003). Let $\overrightarrow{\mathrm{N}_1}$ be equal to ${}_6^0\mathrm{T6} \times \overrightarrow{\mathrm{n}_1}$ and equation 3.8 be arranged as

$$\overrightarrow{l_1} = \overrightarrow{\mathrm{N}_1} - \overrightarrow{P_1} - \overrightarrow{d_1} \tag{3.10}$$

If the vectors and their components are rewritten in matrix form, then

$$\vec{\mathrm{l}}_1 = \begin{bmatrix} \mathrm{N}_{1,x} \\ \mathrm{N}_{1,y} \\ \mathrm{N}_{1,z} \\ 1 \end{bmatrix} - \begin{bmatrix} \mathrm{P}_{1,x} \\ 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} \mathrm{d}_{1,x} \\ \mathrm{d}_{1,y} \\ \mathrm{d}_{1,z} \\ 1 \end{bmatrix}$$

$$\vec{\mathrm{l}}_1 = \begin{bmatrix} \mathrm{N}_{1,x} - \mathrm{P}_{1,x} - \mathrm{d}_{1,x} \\ \mathrm{N}_{1,y} - \mathrm{d}_{1,y} \\ \mathrm{N}_{1,z} - \mathrm{d}_{1,z} \\ 1 \end{bmatrix} \tag{3.11}$$

As put before, the lengths of the links are constant and known; thus, when the right hand side of equation 3.11 is normalized

$$l_1{}^2 = (N_{1,X} - P_{1,X} - d_{1,X})^2 + (N_{1,Y} - d_{1,Y})^2 + (N_{1,Z} - d_{1,Z})^2 \qquad (3.12)$$

Then, when the variable $P_{1,X}$ is separated

$$P_{1,X} = N_{1,X} - d_{1,X} \mp \sqrt{l_1{}^2 - (N_{1,Y} - d_{1,Y})^2 - (N_{1,Z} - d_{1,Z})^2} \qquad (3.13)$$

From equation 3.13, the actuator value $P_{1,X}$ has two solutions (Johannesson 2003).

This chain of calculations will be repeated for the remaining prismatic actuators as well. The schematic description of the vectors that comprise link 3 is in figure 38.

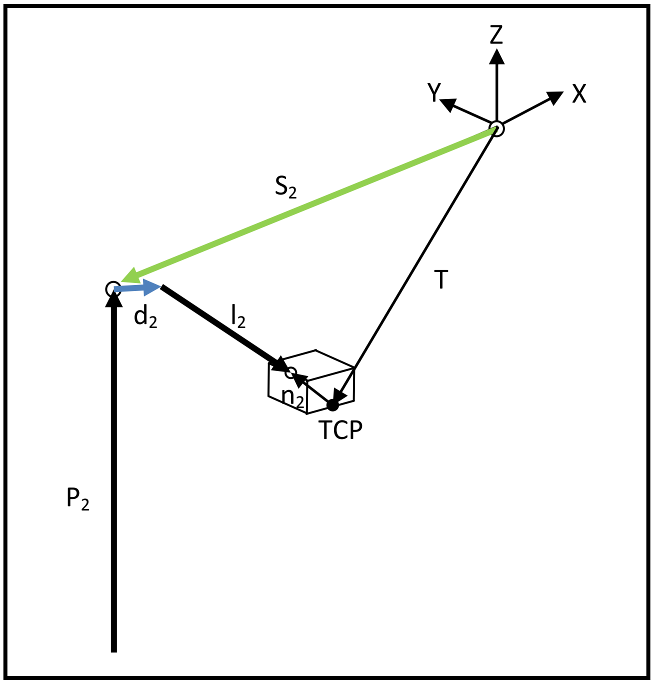


Figure 38: Prismatic joint 2 and the corresponding vectors

For the second actuator, the same type of calculation done for the first will be created. The sum of the vectors presented in figure 38 corresponds to

$$\overrightarrow{S_2} + \overrightarrow{P_2} + \overrightarrow{d_2} + \overrightarrow{l_2} = {}_6^0\mathrm{T6} \times \overrightarrow{\mathrm{n}_2} \qquad (3.14)$$

As in link 1, let the right hand side of equation 3.14 be $\overrightarrow{\mathrm{N}_2}$ and when the left hand side of the equation is separated and represented in matrix form

$$\overrightarrow{l_2} = \overrightarrow{\mathrm{N}_2} - \overrightarrow{S_2} - \overrightarrow{P_2} - \overrightarrow{d_2}$$

$$\vec{l}_2 = \begin{bmatrix} N_{2,x} \\ N_{2,y} \\ N_{2,z} \\ 1 \end{bmatrix} - \begin{bmatrix} S_{2,y} \\ S_{2,y} \\ S_{2,z} \\ 1 \end{bmatrix} - \begin{bmatrix} P_{2,x} \\ 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} d_{2,x} \\ d_{2,y} \\ d_{2,z} \\ 1 \end{bmatrix}$$

$$\vec{l}_2 = \begin{bmatrix} N_{2,x} - S_{2,x} - P_{2,x} - d_{2,x} \\ N_{2,y} - S_{2,y} - d_{2,y} \\ N_{2,z} - S_{2,z} - d_{2,z} \\ 1 \end{bmatrix} \tag{3.15}$$

When both hand sides of equation 3.15 are normalized, then

$$l_2{}^2 = (N_{2,x} - S_{2,x} - P_{2,X} - d_{2,x})^2 + (N_{2,y} - S_{2,y} - d_{2,y})^2 + (N_{2,z} - S_{2,z} - d_{2,z})^2 \tag{3.16}$$

With separation of variables of equation 3.16, the prismatic joint value

$$P_{2,X} = N_{2,x} - S_{2,x} - d_{2,z} \mp \sqrt{l_2{}^2 - (N_{2,y} - S_{2,y} - d_{2,y})^2 - (N_{2,z} - S_{2,z} - d_{2,z})^2} \tag{3.17}$$

As in earlier joint, two solutions exist for $P_{2,X}$ as well (Johannesson 2003). The last actuator is attached to the last link l_6 in the robot. Thus the vectors that constitute this chain can be seen in figure 39.

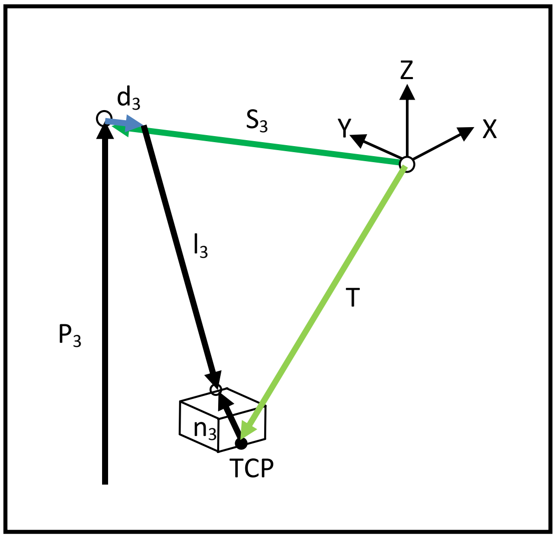


Figure 39: Prismatic joint 3's vectors

Repeating the same idea for other joints, the sum of the vectors for the last prismatic joint will yield

$$\overrightarrow{S_3} + \overrightarrow{P_3} + \overrightarrow{l_3} = {}^0_6\mathrm{T6} \times \overrightarrow{\mathrm{n}_3} \tag{3.18}$$

Let the right hand side of equation 3.18 be $\overrightarrow{\mathrm{N}_3}$ and when the left hand side of the equation is separated and represented in matrix form

$$\overrightarrow{l_3} = \overrightarrow{\mathrm{N}_3} - \overrightarrow{S_3} - \overrightarrow{P_3} - \overrightarrow{d_3}$$

$$\vec{\mathrm{l}}_3 = \begin{bmatrix} \mathrm{N}_{3,\mathrm{x}} \\ \mathrm{N}_{3,\mathrm{y}} \\ \mathrm{N}_{3,\mathrm{z}} \\ 1 \end{bmatrix} - \begin{bmatrix} \mathrm{S}_{3,x} \\ \mathrm{S}_{3,\mathrm{y}} \\ \mathrm{S}_{3,z} \\ 1 \end{bmatrix} - \begin{bmatrix} \mathrm{P}_{3,\mathrm{x}} \\ 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} d_{3,\mathrm{x}} \\ d_{3,y} \\ d_{3,z} \\ 1 \end{bmatrix}$$

$$\vec{\mathrm{l}}_3 = \begin{bmatrix} \mathrm{N}_{3,\mathrm{x}} - \mathrm{S}_{3,\mathrm{x}} - \mathrm{P}_{3,\mathrm{x}} - d_{3,\mathrm{x}} \\ \mathrm{N}_{3,\mathrm{y}} - \mathrm{S}_{3,\mathrm{y}} - d_{3,\mathrm{y}} \\ \mathrm{N}_{3,\mathrm{z}} - \mathrm{S}_{3,\mathrm{z}} - d_{3,\mathrm{z}} \\ 1 \end{bmatrix} \tag{3.19}$$

When both hand sides of equation 3.19 are normalized, then

$$l_3{}^2 = (N_{3,X} - S_{3,X} - P_{3,X} - d_{3,X})^2 + (N_{3,Y} - S_{3,Y} - d_{3,Y})^2 + (N_{3,Z} - S_{3,Z} - d_{3,Z})^2 \tag{3.20}$$

With separation of variables of equation 3.16, the prismatic joint value is

$$P_{3,X} = N_{3,X} - S_{3,X} - d_{3,X} \mp \sqrt{l_3{}^2 - (N_{3,Y} - S_{3,Y} - d_{3,Y})^2 - (N_{3,Z} - S_{3,Z} - d_{3,Z})^2} \tag{3.21}$$

The last joint has two solutions as well as the earlier joints. Thus, in total Gantry-Tau robot has eight different postures for given TCP values. The robot chooses the best configuration/posture in real-time applications whereas in DELMIA V5 users are allowed to specify the posture. Another important point to keep in mind is that the forward kinematics of Gantry-Tau robot also offers various postures for given joint values (Johannesson 2003). Since such situation cannot be defined in DELMIA V5, it is possible to have some errors when a change is made between the postures offered by inverse kinematics. This problem usually changes the coordinates TCP when the current posture is changed (for the forward kinematics calculation, refer to the appendix).

Before proceeding to the creation of C-file, the necessary coordinates for Gantry-Tau is given in table 4.

	S_1	S_2	S_3	d_1	d_2	d_3	n_1	n_2	n_3
X	0	-1100	-2200	-96.569	0	96.569	224.999	0	-182.574
Y	0	700	0	-185	-322.843	0	-240.001	39.705	-80
Z	0	0	0	-400	-173.726	-400	171.568	336.862	213.994

Table 4: Coordinates of the vectors for Gantry-Tau

4.6. THEORY OF GANTRY-TAU ROBOT TO C-FILE

As with previous cases, the variables should be defined. These variables for the given vectors in Table 4 are defined as

```
/***---------------- User code begins here ---------------------------------
***/
//The vectors to define the prismatic joints where Pi[2][0] is the prismatic
joint value in negative direction
P1[0][0] =          0; P1[1][0] =        0; P1[2][0] =        0; P1[3][0] = 1;
P2[0][0] =      -1100; P2[1][0] =      700; P2[2][0] =        0; P2[3][0] = 1;
P3[0][0] =      -2200; P3[1][0] =        0; P3[2][0] =        0; P3[3][0] = 1;

//The constant vectors to define the upper attachment points from prismatic
joint end
d1[0][0] =    -96.569; d1[1][0] =     -185; d1[2][0] =     -400; d1[3][0] = 1;
d2[0][0] =          0; d2[1][0] = -322.843; d2[2][0] = -173.726; d2[3][0] = 1;
d3[0][0] =     96.569; d3[1][0] =        0; d3[2][0] =     -400; d3[3][0] = 1;

//The vectors that connect TCP to lower attachment points.
n1[0][0] =   224.999; n1[1][0] = -240.001;n1[2][0] =  171.568; n1[3][0] = 1;
n2[0][0] =         0; n2[1][0] =   39.705; n2[2][0] = 336.862; n2[3][0] = 1;
n3[0][0] =  -182.574; n3[1][0] =      -80; n3[2][0] = 213.994; n3[3][0] = 1;

//Reference lengths
Lref=1500 ;
Lref2=1499.775;
```

Again TCP matrix will be formed from T6 matrix as following

```
//Importing the current TCP values from Delmia through the T6 matrix and
putting proper context
nx = T6[0][0];
ny = T6[0][1];
nz = T6[0][2];
ox = T6[1][0];
oy = T6[1][1];
oz = T6[1][2];
ax = T6[2][0];
ay = T6[2][1];
az = T6[2][2];
px = T6[3][0];
py = T6[3][1];
pz = T6[3][2];

//The transforming T6 matrix from row vectors form to column vector form
TCP[0][0] =  nx; TCP[0][1] =  ox; TCP[0][2] =  ax; TCP[0][3] = px;
TCP[1][0] =  ny; TCP[1][1] =  oy; TCP[1][2] =  ay; TCP[1][3] = py;
TCP[2][0] =  nz; TCP[2][1] =  oz; TCP[2][2] =  az; TCP[2][3] = pz;
TCP[3][0] =   0;  TCP[3][1] = 0;   TCP[3][2] =  0;  TCP[3][3] = 1;
```

Then, the multiplication of n_i vectors with TCP matrix will be accomplished via

```
//Calculating the current position (in x,y,z in Base coordinates) of each
lower attachment point for each leg by multiplying the transformation
// matrix TCP[4][4] with the vector between the current TCP (the T6 matrix)
and the lower attachmentpoint for each leg

//Calculate upper position on Leg1 (The array L1tCur)
 for (row1 = 0; row1 < 4; row1++) {
        for (col1 = 0; col1 < 1; col1++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner1 = 0; inner1 < 4; inner1++) {
                NT1[row1][0] += TCP[row1][inner1] * n1[inner1][col1];
            }
        }
  }
 //Calculate upper position on Leg2 (The array L2tCur)
    for (row2 = 0; row2 < 4; row2++) {
        for (col2 = 0; col2 < 1; col2++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner2 = 0; inner2 < 4; inner2++) {
                NT2[row2][0] += TCP[row2][inner2] * n2[inner2][col2];
            }
        }
  }
//Calculate upper position on Leg3 (The array L3tCur)
    for (row3 = 0; row3 < 4; row3++) {
        for (col3 = 0; col3 < 1; col3++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner3 = 0; inner3 < 4; inner3++) {
                NT3[row3][0] += TCP[row3][inner3] * n3[inner3][col3];
            }
        }
  }
```

Now, equations 3.13, 3.17, and 3.21 will be directly applied as following

```
//Finding the joint values by using the theory. Multiplication with -1 stems
from the direction of the joints.

J11=-1*(NT1[2][0]-d1[2][0]+ sqrt(pow(Lref,2)-pow((P1[0][0]+d1[0][0]-
NT1[0][0]),2)-pow((P1[1][0]+d1[1][0]-NT1[1][0]),2)));

J12=-1*(NT1[2][0]-d1[2][0]- sqrt(pow(Lref,2)-pow((P1[0][0]+d1[0][0]-
NT1[0][0]),2)-pow((P1[1][0]+d1[1][0]-NT1[1][0]),2)));

J21=-1*(NT2[2][0]-d2[2][0]+ sqrt(pow(Lref,2)-pow((P2[0][0]+d2[0][0]-
NT2[0][0]),2)-pow((P2[1][0]+d2[1][0]-NT2[1][0]),2)));

J22=-1*(NT2[2][0]-d2[2][0]- sqrt(pow(Lref,2)-pow((P2[0][0]+d2[0][0]-
NT2[0][0]),2)-pow((P2[1][0]+d2[1][0]-NT2[1][0]),2)));
```

```
J31=-1*(NT3[2][0]-d3[2][0]+ sqrt(pow(Lref2,2)-pow((P3[0][0]+d3[0][0]-
NT3[0][0]),2)-pow((P3[1][0]+d3[1][0]-NT3[1][0]),2)));
```

```
J32=-1*(NT3[2][0]-d3[2][0]- sqrt(pow(Lref2,2)-pow((P3[0][0]+d3[0][0]-
NT3[0][0]),2)-pow((P3[1][0]+d3[1][0]-NT3[1][0]),2)));
```

Then, the joint values are sent back to solutions matrix for all possible eight postures that can be created with J1, J2 and J3's two different values. The matrix then should be as

```
//Sending the final joint values back to the "solutions"-matrix which is the
input matrix for Delmia.
solutions[0][0] = J11;  solutions[1][0] = J21; solutions[2][0] = J31;
solutions[0][1] = J11;  solutions[1][1] = J21; solutions[2][1] = J32;
solutions[0][2] = J11;  solutions[1][2] = J22; solutions[2][2] = J31;
solutions[0][3] = J11;  solutions[1][3] = J22; solutions[2][3] = J32;
solutions[0][4] = J12;  solutions[1][4] = J21; solutions[2][4] = J31;
solutions[0][5] = J12;  solutions[1][5] = J21; solutions[2][5] = J32;
solutions[0][6] = J12;  solutions[1][6] = J22; solutions[2][6] = J31;
solutions[0][7] = J12;  solutions[1][7] = J22; solutions[2][7] = J32;
```

5. RECOMMENDED COURSE OF ACTION FOR MECHANISM BUILDING IN DELMIA V5 – THE USER-FRIENDLY METHODOLOGY

As the title suggests, this chapter is dedicated to describe the methodology employed to create a mechanism beyond the scope of technical knowledge required for any user to utilize DELMIA V5's *Device Building* module. One of the most important aspects of mechanism building in the respective software is that any user can encounter various types of obstacles. Thus, in the preceding parts of this chapter each step of the designed methodology will be described.

The first step of each device building project in DELMIA V5 is to first investigate the relative theory for inverse and forward kinematics of mechanism. By building this knowledge in advance, the users will not only learn about the inverse kinematics, but also be able to inherit the necessary understanding for the expected behavior of the target mechanism.

The next step, as expected, is to create the forward kinematics of the mechanism in hand. Specifically, each mechanism may require a different approach than the next; however, the very basics of the mechanism building will still remain within the idea that first appoint the fixed part and successively build the mobile parts of the mechanism. When the end-effector is reached, the behavior of the mechanism should be checked via jogging the mechanism. Thus, the schematic methodology will look like for forward kinematics as in figure 40.

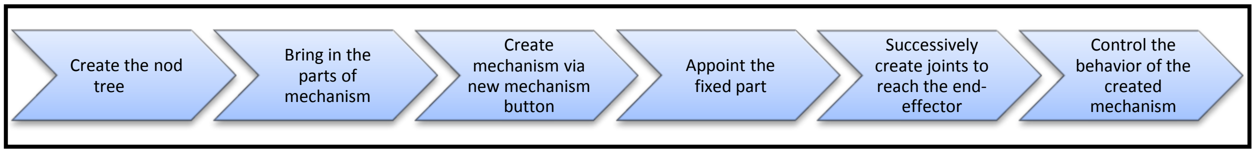

Create the nod tree
Bring in the parts of mechanism
Create mechanism via new mechanism button
Appoint the fixed part
Successively create joints to reach the end-effector
Control the behavior of the created mechanism

Figure 40: Forward kinematics steps

After reaching a correct state in forward kinematics, the users now should analyze the inverse kinematics theory. The available literature may offer various ways to define inverse kinematics; however, since DELMIA V5 offers transformation matrix for the end-effector it would be important to apply the relevant theory with respect to such opportunity. After having defined the relative theory, users should also define the inverse kinematics parameters via the interface offered by DELMIA V5. This wizard/interface walks the users through the necessary parameters, which are well described in the appendices of this thesis work. Thirdly, it is highly-recommended for users to transfer the inverse kinematics theory to a MATLAB function as well, where users will be able to compare their simulation results to those coming from MATLAB. However, it is also important to verify MATLAB functions via using the created mechanism that has only forward kinematics definition. Users can verify these MATLAB functions by simply jogging the mechanism to a certain TCP position and orientation; and when these TCP values are applied to MATLAB, the outcome of the function should match the joint values in DELMIA V5. When these steps are collected and put in a scheme, the result will be as figure 41.

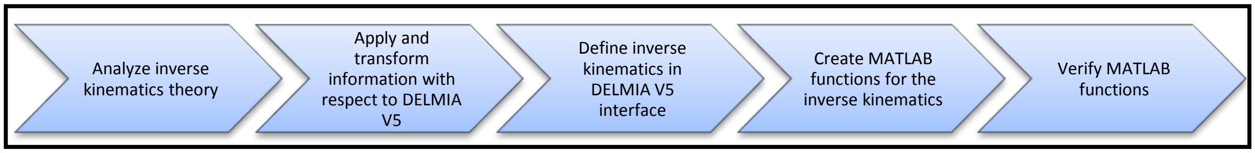

Analyze inverse kinematics theory
Apply and transform information with respect to DELMIA V5
Define inverse kinematics in DELMIA V5 interface
Create MATLAB functions for the inverse kinematics
Verify MATLAB functions

Figure 41: Inverse kinematics steps

After the parameter definition and verification of MATLAB functions, users are guided to create the C-file with analyzed and applied inverse kinematics description. Firstly, users should appoint the administrative commands of the C-file, which are plainly described in the theory section. Then, users should transfer the inverse kinematics theory to the file and compile it. The final step before proceeding to the testing phase is the compilation of the C-file. This phase requires the use of a compilation tool. Such necessity is discussed in section 3.3.3 and APPENDIX F: COMPILATION OF C-FILES. Users are not expected to have any problems while using this step of the methodology employed for this thesis work. So the steps then will be as in figure 42.

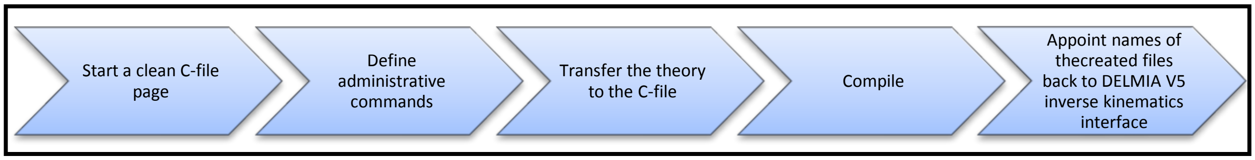

Start a clean C-file page
Define administrative commands
Transfer the theory to the C-file
Compile
Appoint names of thecreated files back to DELMIA V5 inverse kinematics interface

Figure 42: C-file creation

After the compilation, debugging phase of the mechanism should be done. During the compilation, it is possible for layman users to encounter with syntax or semantic errors. Thus, these errors can easily be spotted on the debugging window. If the compilation in the first place is successful, the accuracy of the calculations should be verified. Hence, it is recommended for users to move the mechanism with inverse kinematics definition to a certain position and orientation; and compare the results to those coming from MATLAB for the same TCP values. If there is a mismatch between the results, the source of error should be spotted and rectified. When the rectification is done, the new C-file should be recompiled. And this circle should continue until it is proven that the mechanism is fully functional and free of errors. When these steps are grouped in a schematic description, the result will be as in figure 43.

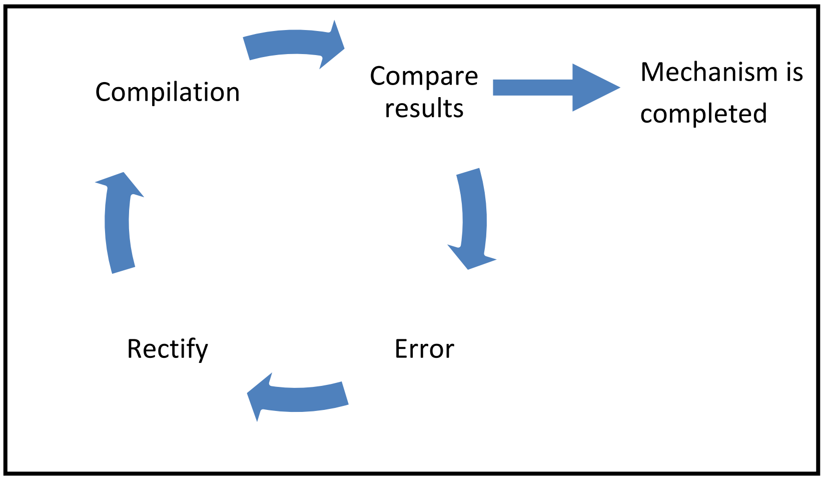


Figure 43: Compilation and debugging

Finally, when all the steps so far described are to be redefined in a complete tree like concept, the resulting figure will be as in figure 44; and the methodology applied to the robots of this thesis work can be seen in figure 45.

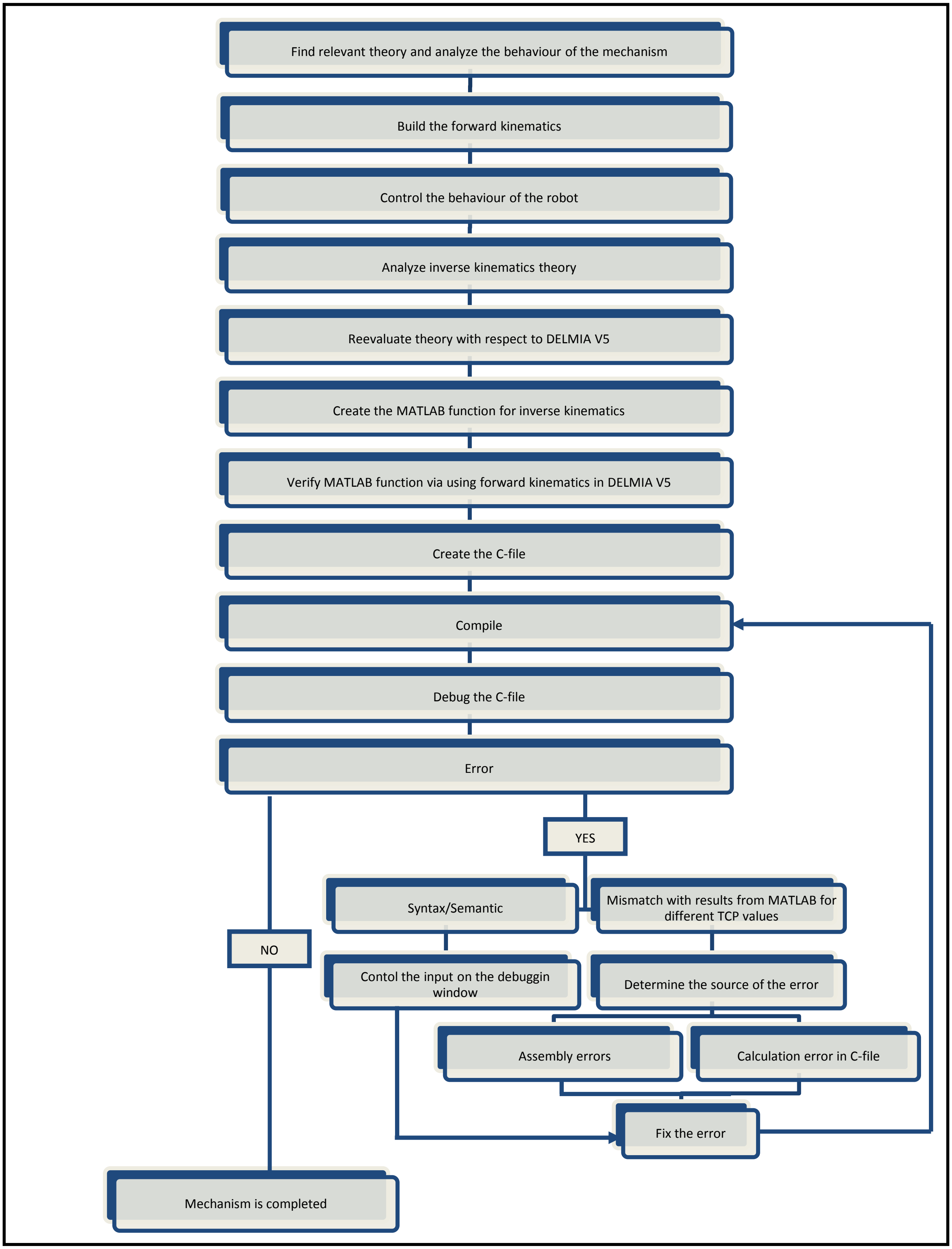

Find relevant theory and analyze the behaviour of the mechanism
Build the forward kinematics
Control the behaviour of the robot
Analyze inverse kinematics theory
Reevaluate theory with respect to DELMIA V5
Create the MATLAB function for inverse kinematics
Verify MATLAB function via using forward kinematics in DELMIA V5
Create the C-file
Compile
Debug the C-file
Error
YES
NO
Syntax/Semantic
Mismatch with results from MATLAB for different TCP values
Contol the input on the debuggin window
Determine the source of the error
Assembly errors
Calculation error in C-file
Fix the error
Mechanism is completed

Figure 44: The recommended course of action

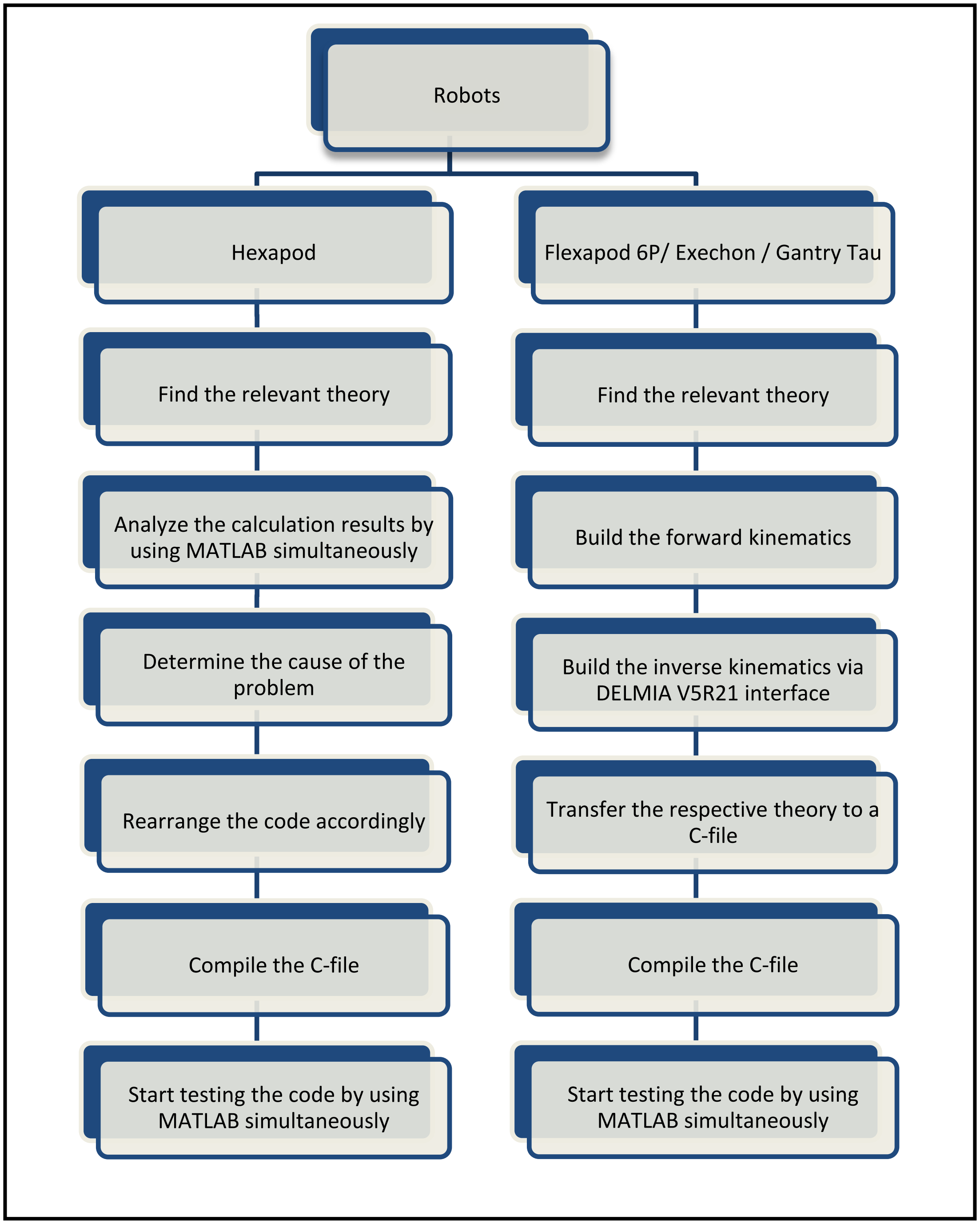

Robots
Hexapod
Flexapod 6P/ Exechon / Gantry Tau
Find the relevant theory
Find the relevant theory
Analyze the calculation results by using MATLAB simultaneously
Build the forward kinematics
Determine the cause of the problem
Build the inverse kinematics via DELMIA V5R21 interface
Rearrange the code accordingly
Transfer the respective theory to a C-file
Compile the C-file
Compile the C-file
Start testing the code by using MATLAB simultaneously
Start testing the code by using MATLAB simultaneously

Figure 45: Application of methodology to respective robots

6. RESULTS AND DISCUSSION

The aftermath of this thesis work is fully functional Hexapod, Flexapod 6P, Exechon and Gantry-Tau robots along with a methodology enabling the simulation of most parallel kinematics structures in DELMIA V5 by using a C-file for inverse kinematics.

The methodology developed for this thesis work can be used by any layman users for the purpose of simulating advanced mechanisms in not only DELMIA V5 but also any other simulation environment that supports the use of inverse kinematics via a C-file. However, it is very likely to encounter some obstacles at some points of the methodology that cannot be classified and illustrated easily in the methodology.

One of these stages that problems are most likely to occur is whilst the creation of forward kinematics. During the assembly, it is observed that the creation of joints may not be possible due to the fact that the mechanism sometimes becomes over-constrained. The reason of such a case is usually resulting from the fact that the algorithm behind DELMIA V5's forward kinematics is not able to find a point in the working space to create the necessary constraints for the joint. In order to overcome such a problem, users can easily drag/rotate the respective part to a point that is close to the center of prospective joint. Usually, the relocation of these parts results in the successful assembly; however, the outcome may differ from what users expect. Even if the over-constraint case did not happen, the algorithm described above can cause unexpected problems such as the case in which the mobile parts of the robot are diving through each other and the end-effector is at some point and orientation other than home position. Such a case is illustrated in figure 46.

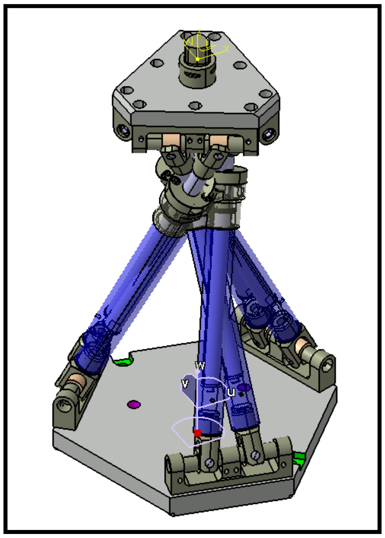


Figure 46: Parts diving each other

Since these problems are randomly occurring and their reasons are somewhat ambiguous, they have not been classified in the methodology. However, this situation clearly shows that the methodology can further be elevated in terms of forward kinematics to decipher all the dynamics of mechanism creation in DELMIA V5.

Another step that is possible for users to run into problems is the inverse kinematics creation. In this step, the methodology reduces problems into two categories – syntax/semantic errors and calculation mistakes. In this thesis work all the errors that occurred during this phase were related to either syntax/semantic or calculation. However, this situation still can be open-ended, meaning users may have to overcome other problems if they need to change the parameters in the DELMIA V5's interface for inverse kinematics or they might suffer from the incapability of C-language when very advanced calculations must be made.

Having described the results of using the methodology, it is also important to represent the key findings with created robots. Consequently, the hexapod robot was the first to be analyzed as it was handed over from the previous owner of the project, Torbjörn Jakobsson. Thus, its forward kinematics was built in advance by him. On the other hand, Flexapod 6P – which shares the same theory with hexapod – was completely built from already available CAD-parts. The theory for inverse kinematics of these robots was completely adopted and it proved to be completely sufficient for the simulation to be used in real-life applications. The error rate of joint values observed throughout the testing phase was approximately in the span of ±0.0004 mm with respect to the output coming from MATLAB functions – the reason of which is presumed to be related to C-language bitwise operations.

The Exechon robot is simulated with only parallel structure, meaning the wrist attached after the mobile platform was not included in inverse kinematics. The reason behind this choice stemmed from the fact that available published theory regarding the inverse kinematics of hybrid structure (parallel and serial attached to each other successively) was not available at the time of this thesis work carried out. In addition, due to its limitations in the kinematics structure Exechon robot is not responding accurately to the commands given by DELMIA V5's Cartesian tab – in which rotational and translational motion cannot happen simultaneously unless a separate tag for the target TCP is created. Particularly, when pure translational motion is performed DELMIA V5 first calculates the necessary joint values for pure translation; and for these joint values Exechon reaches the closest possible point in its workspace. In table 5, such situation is disclosed for 100 mm of translation in X-direction. When the motion is performed DELMIA V5 takes the robot to the nearest point $[95.7935, 49.4846, -1252.5967]^T$ via translational and rotational motion. When analyzed further, these limitations in the capability of the robot motion are stemming from the fact that the mobile platform is only capable of doing rotations about X and Y axes and pure translation in Z-direction (Bi 2011).

	Intended TCP Coordinates	**Intended TCP orientation (deg)**	**Actual TCP Coordinates**	**Actual TCP orientati on (deg)**	**Joint values for intended TCP coordinates from MATLAB function (mm)**	**Joint values for actual TCP coordinates from debugging window (mm)**	**Joint values for actual TCP coordinates from MATLAB function(mm)**
X	100	0	95.7935	-0.024	-24.8915	-24.8915	-24.8920
Y	50	0	49.4846	-4.372	36.1332	36.1332	36.1329
Z	-1250	0	-1252.5967	0.002	5.6258	5.6258	5.6257

Table 5: Exechon's forward kinematics results

The Gantry-Tau robot has been simulated and proved to be mostly functional in simulation environment. The theory of inverse kinematics for this robot was easily integrated into a C-file except for switching between the eight different postures that the inverse kinematics of Gantry-Tau robot offers. Specifically, for a given TCP values, the inverse kinematics calculates eight different postures. When changing the posture, it was observed that TCP values changed even though they should have remained the same as in the earlier posture. The reason for this problem was resulting from the fact that when joint values are fed back to DELMIA V5, forward kinematics of the robot recalculates the TCP coordinates and sends these values back to the transformation matrix. However, the forward kinematics calculation of Gantry-Tau results in multiple postures as for its inverse kinematics (see APPENDIX E: GANTRY-TAU ROBOT for forward kinematics calculation). Since DELMIA V5 does not include this option to choose between postures in forward kinematics calculation, the software randomly chooses an available posture and sends different TCP coordinates to C-file than the values of intended TCP. To illustrate, when Gantry-Tau robot is jogged to $[-1200, -750, -1900]^T$ mm and forced to switch between the postures resulting from the inverse kinematics calculation, the next posture causes TCP to move to another point $[-725.0365, 714.1958, -2274.3482]^T$ although it is intended to remain in its former position as seen in table 6. As Johannesson (2003) explains this situation is stemming from the fact that when posture 2's joint values are applied to forward kinematics, it produces multiple TCP values; and as DELMIA V5 is not offering this choice between TCPs, it is randomly appointing one TCP location for the robot. On the other hand, it was also observed that this situation does not happen for some postures as well. Hence, it is enough to make the point that if users choose to remain in the same posture as the TCP propagates, they would be able to reach a somewhat stable simulation environment for Gantry-Tau robot.

	Intended TCP coordinates	Joint values for the intended coordinates from debugging window	Joint values for Posture 1 at intended TCP location	Joint values for Posture 2 when switching postures	New TCP coordinates when switched to posture 2	Joint values for posture 2's TCP coordinates from MATLAB function (mm)
X	-1200	417.2158/2239.6689	417.2158	417.2158	-725.0365	417.2157
Y	-750	361.0962/2417.7485	361.0962	361.0962	714.1958	361.0964
Z	-1900	265.8219/2306.2108	265.8219	2306.2108	-2274.3482	2306.2107

Table 6: Gantry-Tau's inverse kinematics results

This thesis work shows that there is still much to be done to reach a complete robust simulation of parallel kinematics mechanisms in DELMIA V5 since the forward kinematics of these structures requires more delicate care than of serial robots. Thus, an interface that calculates the forward kinematics of parallel structures would pave the way for DELMIA V5 to enhance its simulation ability over advanced mechanisms. Also, further exploration of the assembly creation algorithm and input parameters of inverse kinematics can lay the foundations for other opportunities in the field of mechanism creation in DELMIA V5.

7. CONCLUSION

In this thesis work, a methodology to create advanced mechanisms in DELMIA V5 has been developed and by using this methodology hexapod, Flexapod 6P, Exechon and Gantry-Tau robots have been simulated. The methodology covers the complete steps that range from mechanism assembly to the application of inverse kinematics theory to DELMIA V5. The compilation and environment set-up along with the testing procedure have also been methodically defined. Therefore, with this methodology layman users would be able to create any type of mechanism in DELMIA V5 or similar environments in which inverse kinematics is defined with a C-file. In addition, a how-to style documentation for layman users have been created in which users can find the steps of how to implement the methodology to respective robots. The simulation of hexapod and Flexapod 6P robots have proved to be completely functional whereas the simulation of Exechon and Gantry-Tau robots still needs some improvements due to the reasons stated in the result section.

REFERENCES

Bi, Z. and Jin, Y., 2011. Kinematic modeling of Exechon parallel kinematic machine. *Robotics and Computer-Integrated Manufacturing*, 27(1), pp.186-193.

Craig, J., 2005. *Introduction to robotics*. 1st ed. Upper Saddle River, N.J.: Pearson/Prentice Hall.

Hartenberg, R. and Denavit, J., 1964. *Kinematic synthesis of linkages*. 1st ed. New York: McGraw-Hill.

Jazar, R., 2010. *Theory of applied robotics*. 1st ed. New York: Springer.

Ji, P. and Wu, H., 2001. A closed-form forward kinematics solution for the 6-6 p Stewart platform. *IEEE Transactions on Robotics and Automation*, 17(4), pp.522-526.

Johannesson, L., Berbyuk, V. and Brogårdh, T., 2003. Gantry-Tau A New Three Degrees of Freedom *Parallel Kinematic Robot. Proceedings of the Mekatronikmöte2003*, August 27-28, 2003, Göteborg, Sweden. pp.1-6

Locomachs.eu, (2014). LOCOMACHS - LOw COst Manufacturing and Assembly of Composite and Hybrid Structures - Welcome to the official website of the LOCOMACHS project!. [online] Available at: http://www.locomachs.eu/ (2014-06-10).

Yang, J. and Geng, Z., 1998. Closed form forward kinematics solution to a class of hexapod robots. *IEEE Transactions on Robotics and Automation*, 14(3), pp.503-508.

Zoppi, M., Zlatanov, D. and Molfino, R., 2010. Kinematics analysis of the Exechon tripod. *Proceedings of the ASME 2010 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, August 15-18, 2010, Montreal, Quebec, Canada. pp.1381-1388.

APPENDICES

The appendices section is structured as how-to style documentation where layman users can easily follow steps to build the respective robots described in the main part of this work without having prerequisite knowledge. In appendix A, environment set-up is clarified. In appendix B, C, D and E complete forward and inverse kinematics of the respective robots are given. In appendix F compilation of a C-file and final arrangements for simulation are depicted. The respective MATLAB functions are given in appendix G.

APPENDIX A: HOW TO SET UP ENVIRONMENT FOR FORWARD AND INVERSE KINEMATICS

- First, debugging window should be arranged as following. Right-click on DELMIA V5 and click properties.

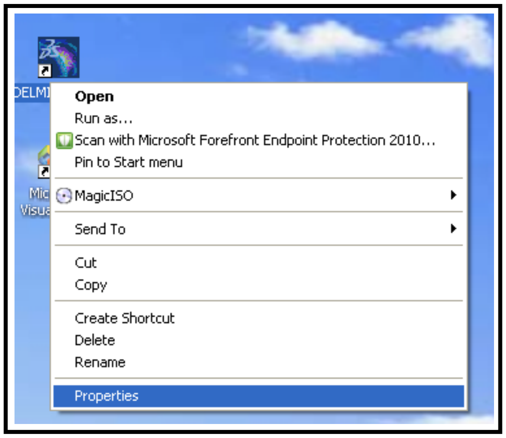
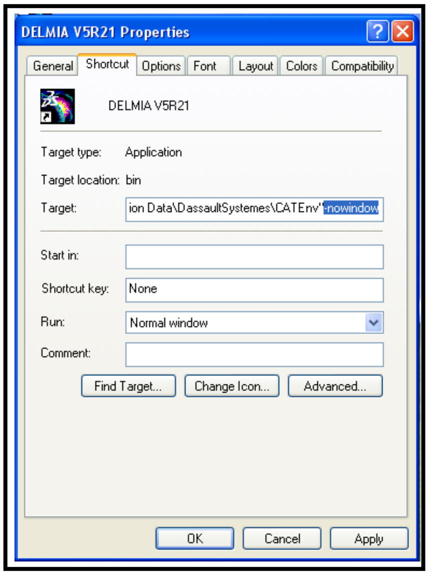


Figure A.1. Properties tab

- On Shortcut Tab, remove the "-nowindow" section from the target directory. Then click '*OK*'.



Target type: Application

Target location: bin

Target: ion Data\DassaultSystemes\CATEnv\-nowindow

Start in:

Shortcut key: None

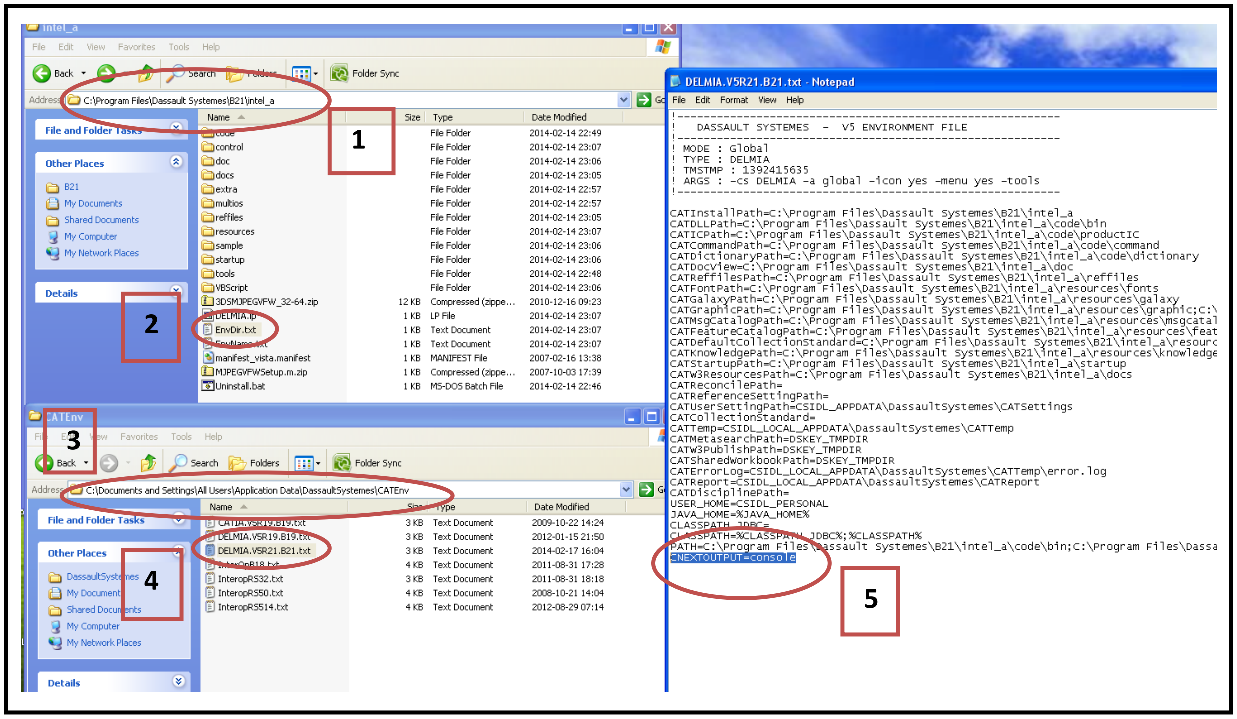
Run: Normal window

Comment:

Figure A.2. Properties tab-nowindow

- Now, an environment variable should be created for DELMIA V5.
  a) Go to directory "C:\Documents and Settings\All Users\Application Data\DassaultSystemes\CATEnv"
  b) If that directory doesn't work, go to "C:\Program Files\Dassault Systemes\B21\intel_a" and open EnvDir.txt file and see the directory of the folder for environment files.
  c) In the *CATEnv* folder, open *DELMIA.V5.B21.txt* file and at the end of the text file enter the following command

CNEXTOUTPUT=console



C:\Program Files\Dassault Systemes\B21\intel_a

Name	Size	Type	Date Modified
code		File Folder	2014-02-14 22:49
control		File Folder	2014-02-14 23:07
doc		File Folder	2014-02-14 23:06
docs		File Folder	2014-02-14 23:05
extra		File Folder	2014-02-14 22:57
multios		File Folder	2014-02-14 22:57
reffiles		File Folder	2014-02-14 23:05
resources		File Folder	2014-02-14 23:07
sample		File Folder	2014-02-14 23:06
startup		File Folder	2014-02-14 23:06
tools		File Folder	2014-02-14 22:48
VBScript		File Folder	2014-02-14 23:06
3DSMJPEGVFW_32-64.zip	12 KB	Compressed (zippe…	2010-12-16 09:23
DELMIA.lp	1 KB	LP File	2014-02-14 23:07
EnvDir.txt	1 KB	Text Document	2014-02-14 23:07
EnvName.txt	1 KB	Text Document	2014-02-14 23:07
manifest_vista.manifest	1 KB	MANIFEST File	2007-02-16 13:38
MJPEGVFWSetup.m.zip	1 KB	Compressed (zippe…	2007-10-03 17:39
Uninstall.bat	1 KB	MS-DOS Batch File	2014-02-14 22:46

C:\Documents and Settings\All Users\Application Data\DassaultSystemes\CATEnv

Name	Size	Type	Date Modified
CATIA.V5R19.B19.txt	3 KB	Text Document	2009-10-22 14:24
DELMIA.V5R19.B19.txt	3 KB	Text Document	2012-01-15 21:50
DELMIA.V5R21.B21.txt	3 KB	Text Document	2014-02-17 16:04
InterOpB18.txt	4 KB	Text Document	2011-08-31 17:28
InteropRS32.txt	3 KB	Text Document	2011-08-31 18:18
InteropRS50.txt	4 KB	Text Document	2008-10-21 14:04
InteropRS514.txt	4 KB	Text Document	2012-08-29 07:14

DELMIA.V5R21.B21.txt - Notepad

```
!-----------------------------------------------------------
!   DASSAULT SYSTEMES  -  V5 ENVIRONMENT FILE
!-----------------------------------------------------------
! MODE : Global
! TYPE : DELMIA
! TMSTMP : 1392415635
! ARGS : -cs DELMIA -a global -icon yes -menu yes -tools
!-----------------------------------------------------------

CATInstallPath=C:\Program Files\Dassault Systemes\B21\intel_a
CATDLLPath=C:\Program Files\Dassault Systemes\B21\intel_a\code\bin
CATICPath=C:\Program Files\Dassault Systemes\B21\intel_a\code\productIC
CATCommandPath=C:\Program Files\Dassault Systemes\B21\intel_a\code\command
CATDictionaryPath=C:\Program Files\Dassault Systemes\B21\intel_a\code\dictionary
CATDocView=C:\Program Files\Dassault Systemes\B21\intel_a\doc
CATReffilesPath=C:\Program Files\Dassault Systemes\B21\intel_a\reffiles
CATFontPath=C:\Program Files\Dassault Systemes\B21\intel_a\resources\fonts
CATGalaxyPath=C:\Program Files\Dassault Systemes\B21\intel_a\resources\galaxy
CATGraphicPath=C:\Program Files\Dassault Systemes\B21\intel_a\resources\graphic;C:\
CATMsgCatalogPath=C:\Program Files\Dassault Systemes\B21\intel_a\resources\msgcatal
CATFeatureCatalogPath=C:\Program Files\Dassault Systemes\B21\intel_a\resources\feat
CATDefaultCollectionStandard=C:\Program Files\Dassault Systemes\B21\intel_a\resourc
CATKnowledgePath=C:\Program Files\Dassault Systemes\B21\intel_a\resources\knowledge
CATStartupPath=C:\Program Files\Dassault Systemes\B21\intel_a\startup
CATW3ResourcesPath=C:\Program Files\Dassault Systemes\B21\intel_a\docs
CATReconcilePath=
CATReferenceSettingPath=
CATUserSettingPath=CSIDL_APPDATA\DassaultSystemes\CATSettings
CATCollectionStandard=
CATTemp=CSIDL_LOCAL_APPDATA\DassaultSystemes\CATTemp
CATMetasearchPath=DSKEY_TMPDIR
CATW3PublishPath=DSKEY_TMPDIR
CATSharedWorkbookPath=DSKEY_TMPDIR
CATErrorLog=CSIDL_LOCAL_APPDATA\DassaultSystemes\CATTemp\error.log
CATReport=CSIDL_LOCAL_APPDATA\DassaultSystemes\CATReport
CATDisciplinePath=
USER_HOME=CSIDL_PERSONAL
JAVA_HOME=%JAVA_HOME%
CLASSPATH_JDBC=
CLASSPATH=%CLASSPATH_JDBC%;%CLASSPATH%
PATH=C:\Program Files\Dassault Systemes\B21\intel_a\code\bin;C:\Program Files\Dassa
CNEXTOUTPUT=console
```

Figure A.3. Opening debugging window

- When the text file is saved and closed, the environment set-up is completed.

APPENDIX B: BUILDING OF FLEXAPOD 6P

Building of the mechanism is going to start with creation of mechanism which builds the forward kinematics automatically. Then inverse kinematics arrangement will be made. It is important to notify here as well that only Flexapod 6P will be built in forward kinematics since hexapod was already built by the previous owner, Torbjörn Jakobsson.

MECHANISM CREATION OF FLEXAPOD 6P

- First step in the creation is to create the new elements under the node tree. This is done by *New Component* command under *Insert* menu.

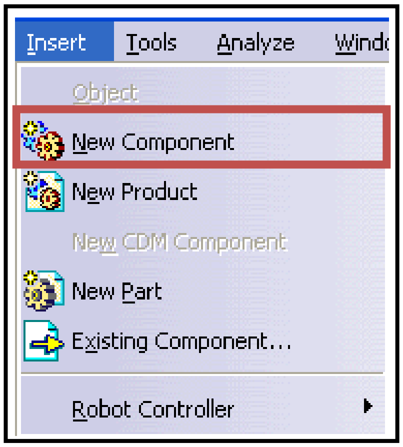
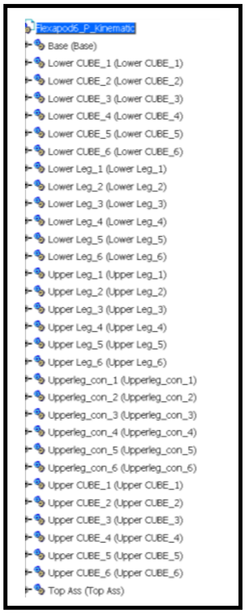


Figure B.1. Opening debugging window

- Create new components under the main node tree as the number of components that the Flexapod 6P has. In this case, 32 new components are required. Then name the components accordingly. The result should look like



```
Flexapod6_P_Kinematic
  Base (Base)
  Lower CUBE_1 (Lower CUBE_1)
  Lower CUBE_2 (Lower CUBE_2)
  Lower CUBE_3 (Lower CUBE_3)
  Lower CUBE_4 (Lower CUBE_4)
  Lower CUBE_5 (Lower CUBE_5)
  Lower CUBE_6 (Lower CUBE_6)
  Lower Leg_1 (Lower Leg_1)
  Lower Leg_2 (Lower Leg_2)
  Lower Leg_3 (Lower Leg_3)
  Lower Leg_4 (Lower Leg_4)
  Lower Leg_5 (Lower Leg_5)
  Lower Leg_6 (Lower Leg_6)
  Upper Leg_1 (Upper Leg_1)
  Upper Leg_2 (Upper Leg_2)
  Upper Leg_3 (Upper Leg_3)
  Upper Leg_4 (Upper Leg_4)
  Upper Leg_5 (Upper Leg_5)
  Upper Leg_6 (Upper Leg_6)
  Upperleg_con_1 (Upperleg_con_1)
  Upperleg_con_2 (Upperleg_con_2)
  Upperleg_con_3 (Upperleg_con_3)
  Upperleg_con_4 (Upperleg_con_4)
  Upperleg_con_5 (Upperleg_con_5)
  Upperleg_con_6 (Upperleg_con_6)
  Upper CUBE_1 (Upper CUBE_1)
  Upper CUBE_2 (Upper CUBE_2)
  Upper CUBE_3 (Upper CUBE_3)
  Upper CUBE_4 (Upper CUBE_4)
  Upper CUBE_5 (Upper CUBE_5)
  Upper CUBE_6 (Upper CUBE_6)
  Top Ass (Top Ass)
```

Figure B.2. The nod-tree for Flexapod 6P

- These 32 components are
  a) Base platform
  b) 12 cubes
  c) 6 lower leg parts
  d) 6 upper leg parts
  e) 6 upper leg connections
  f) Mobile platform

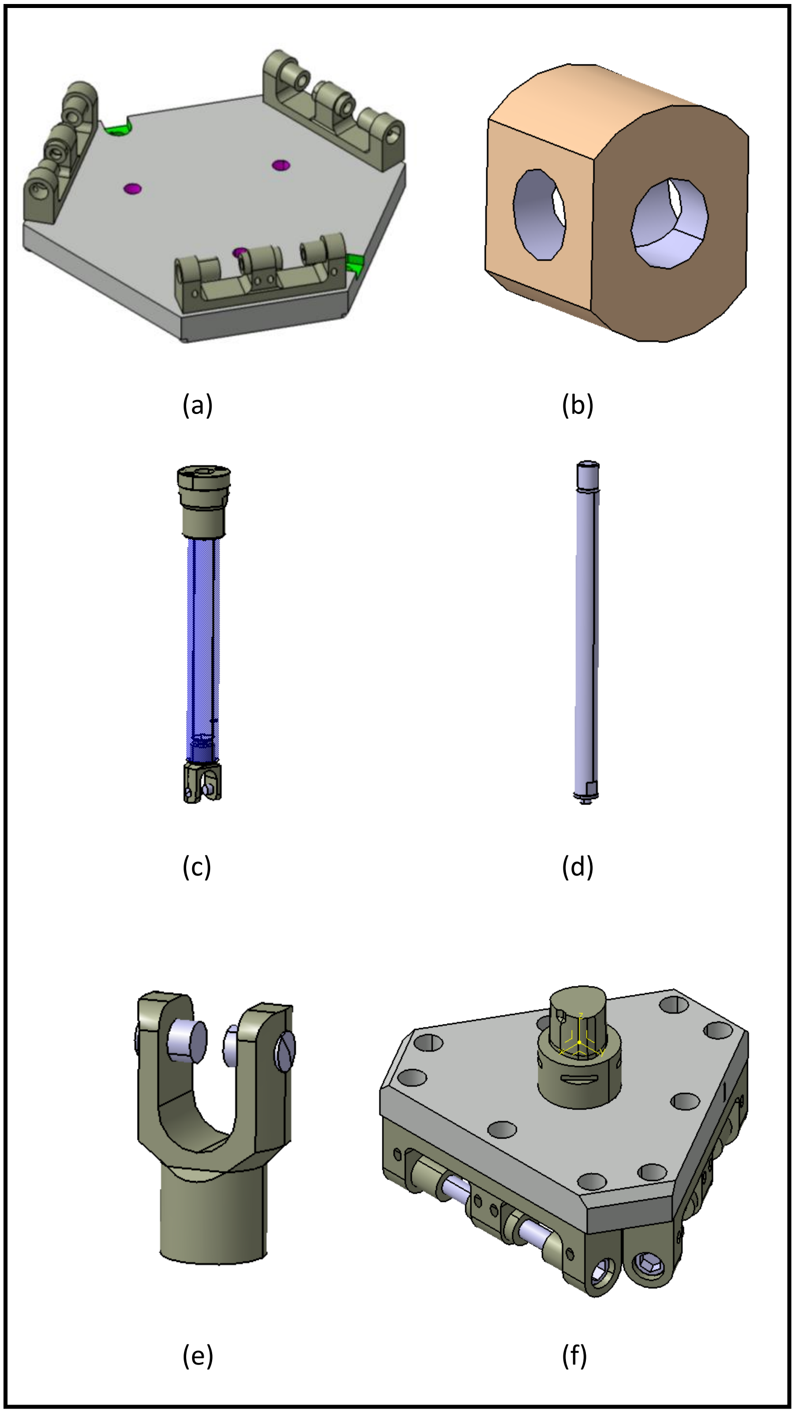

(a) (b) (c) (d) (e) (f)

Figure B.3. The components of Flexapod 6P

- Now, importing the cad or cgr parts into is of topic. To do that, click on the newly created component under the node tree, and insert the respective part via *Insert – Existing Component*.

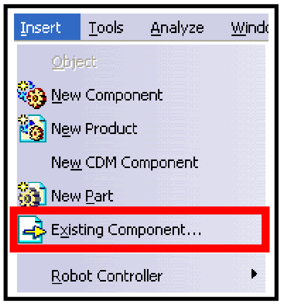
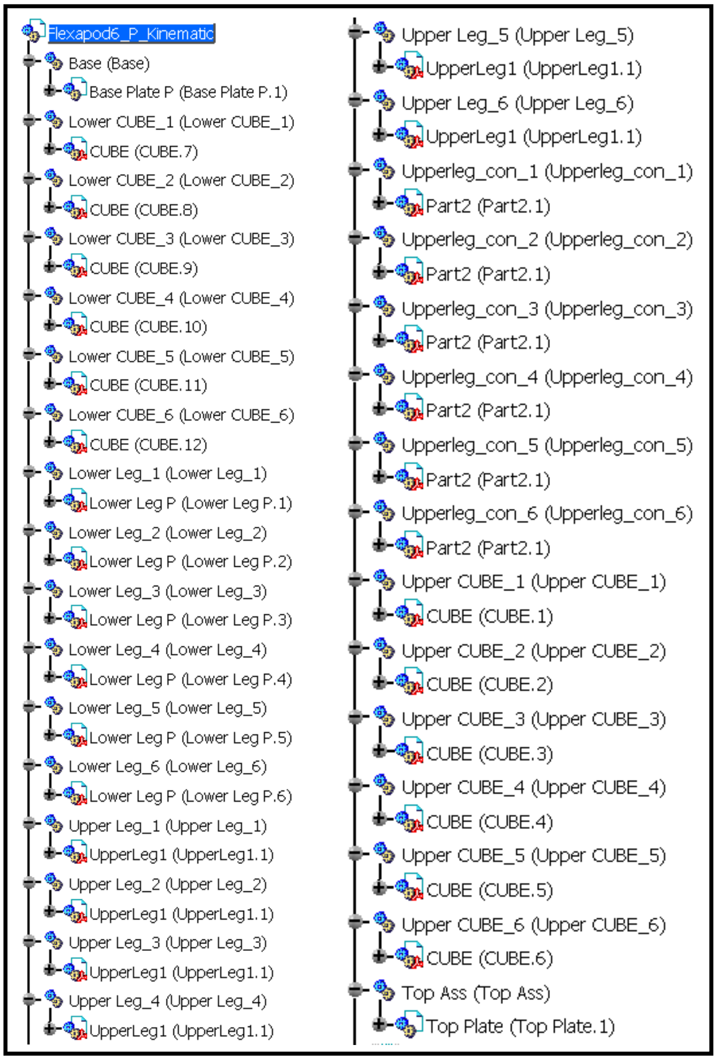


Figure B.4. Inserting CAD-models via Existing Component

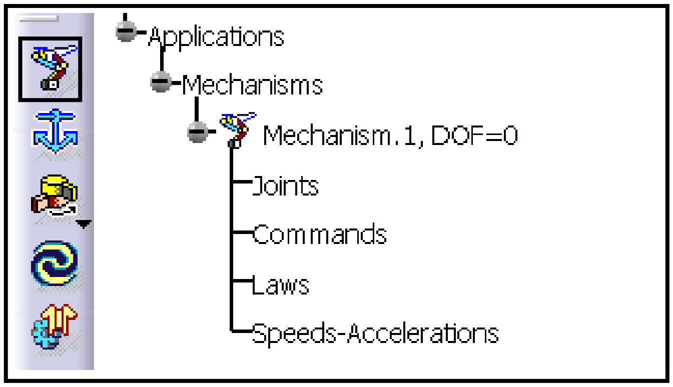
- Now, repeat this procedure of inserting components for the remaining 31 parts. The resulting node tree should look like



Flexapod6_P_Kinematic
- Base (Base)
  - Base Plate P (Base Plate P.1)
- Lower CUBE_1 (Lower CUBE_1)
  - CUBE (CUBE.7)
- Lower CUBE_2 (Lower CUBE_2)
  - CUBE (CUBE.8)
- Lower CUBE_3 (Lower CUBE_3)
  - CUBE (CUBE.9)
- Lower CUBE_4 (Lower CUBE_4)
  - CUBE (CUBE.10)
- Lower CUBE_5 (Lower CUBE_5)
  - CUBE (CUBE.11)
- Lower CUBE_6 (Lower CUBE_6)
  - CUBE (CUBE.12)
- Lower Leg_1 (Lower Leg_1)
  - Lower Leg P (Lower Leg P.1)
- Lower Leg_2 (Lower Leg_2)
  - Lower Leg P (Lower Leg P.2)
- Lower Leg_3 (Lower Leg_3)
  - Lower Leg P (Lower Leg P.3)
- Lower Leg_4 (Lower Leg_4)
  - Lower Leg P (Lower Leg P.4)
- Lower Leg_5 (Lower Leg_5)
  - Lower Leg P (Lower Leg P.5)
- Lower Leg_6 (Lower Leg_6)
  - Lower Leg P (Lower Leg P.6)
- Upper Leg_1 (Upper Leg_1)
  - UpperLeg1 (UpperLeg1.1)
- Upper Leg_2 (Upper Leg_2)
  - UpperLeg1 (UpperLeg1.1)
- Upper Leg_3 (Upper Leg_3)
  - UpperLeg1 (UpperLeg1.1)
- Upper Leg_4 (Upper Leg_4)
  - UpperLeg1 (UpperLeg1.1)
- Upper Leg_5 (Upper Leg_5)
  - UpperLeg1 (UpperLeg1.1)
- Upper Leg_6 (Upper Leg_6)
  - UpperLeg1 (UpperLeg1.1)
- Upperleg_con_1 (Upperleg_con_1)
  - Part2 (Part2.1)
- Upperleg_con_2 (Upperleg_con_2)
  - Part2 (Part2.1)
- Upperleg_con_3 (Upperleg_con_3)
  - Part2 (Part2.1)
- Upperleg_con_4 (Upperleg_con_4)
  - Part2 (Part2.1)
- Upperleg_con_5 (Upperleg_con_5)
  - Part2 (Part2.1)
- Upperleg_con_6 (Upperleg_con_6)
  - Part2 (Part2.1)
- Upper CUBE_1 (Upper CUBE_1)
  - CUBE (CUBE.1)
- Upper CUBE_2 (Upper CUBE_2)
  - CUBE (CUBE.2)
- Upper CUBE_3 (Upper CUBE_3)
  - CUBE (CUBE.3)
- Upper CUBE_4 (Upper CUBE_4)
  - CUBE (CUBE.4)
- Upper CUBE_5 (Upper CUBE_5)
  - CUBE (CUBE.5)
- Upper CUBE_6 (Upper CUBE_6)
  - CUBE (CUBE.6)
- Top Ass (Top Ass)
  - Top Plate (Top Plate.1)

Figure B.5. The components of Flexapod 6P with CAD-models

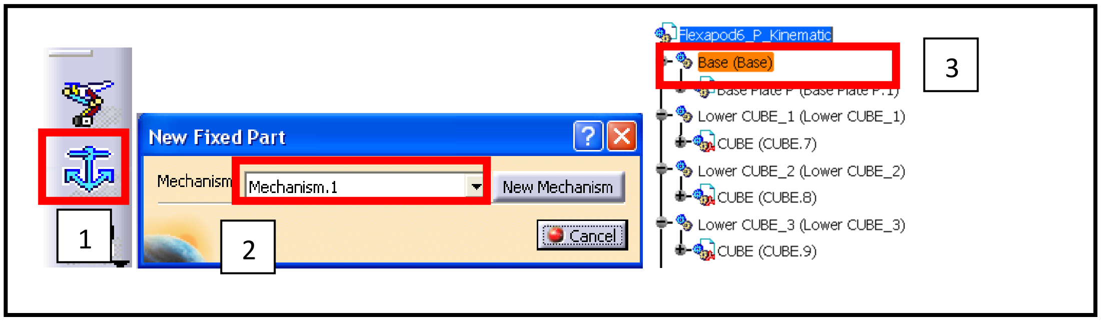
- After bringing all necessary components, a new mechanism should be created first. Click *New Mechanism* and the result will be a new section under the node tree.



- Applications
  - Mechanisms
    - Mechanism.1, DOF=0
      - Joints
      - Commands
      - Laws
      - Speeds-Accelerations

Figure B.6. Creating new mechanism

- Now, set the fixed part of the mechanism, which is the *Base* by using the *Fixed Part* button and then selecting the component named *Base*.

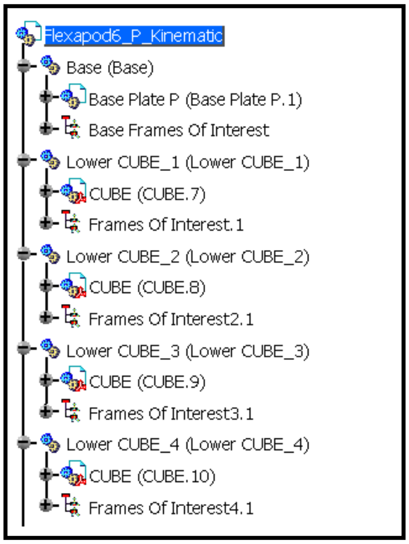


Mechanism: Mechanism.1

- Flexapod6_P_Kinematic
  - Base (Base)
    - Base Plate P (Base Plate P.1)
  - Lower CUBE_1 (Lower CUBE_1)
    - CUBE (CUBE.7)
  - Lower CUBE_2 (Lower CUBE_2)
    - CUBE (CUBE.8)
  - Lower CUBE_3 (Lower CUBE_3)
    - CUBE (CUBE.9)

Figure B.7. Defining fixed-part

- At this moment, Frames of Interests will be created as folders under the node tree. For each component, create the folder Frames of Interests by clicking button. The result should look like this for each component. Name the folders as seen fit.



- Flexapod6_P_Kinematic
  - Base (Base)
    - Base Plate P (Base Plate P.1)
    - Base Frames Of Interest
  - Lower CUBE_1 (Lower CUBE_1)
    - CUBE (CUBE.7)
    - Frames Of Interest.1
  - Lower CUBE_2 (Lower CUBE_2)
    - CUBE (CUBE.8)
    - Frames Of Interest2.1
  - Lower CUBE_3 (Lower CUBE_3)
    - CUBE (CUBE.9)
    - Frames Of Interest3.1
  - Lower CUBE_4 (Lower CUBE_4)
    - CUBE (CUBE.10)
    - Frames Of Interest4.1

Figure B.8. Creating FOI folders

- Now, the first mechanism will be created. Start with appointing FOI to the *Base* part as in the figure. Click button and select the FOI folder under the *Base*. A new menu will appear and it will ask for the location and orientation of the new FOI. Select the center of the prospective joint and make sure that rotation is about the Z-axis of FOI. The steps should look like

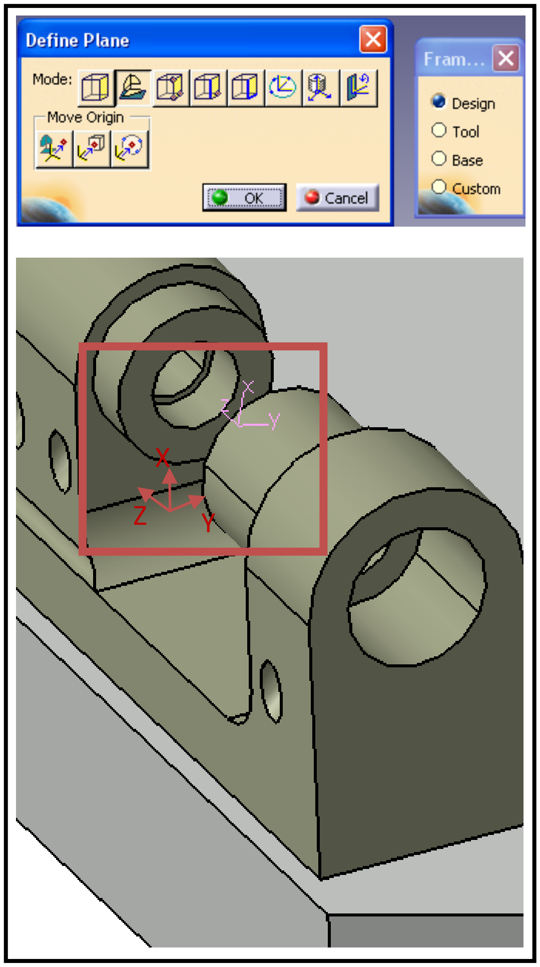


Figure B.9. Creating FOIs in the fixed part

- Now, a FOI will be appointed to the corresponding *Cube* part. Same procedure for the Base part should be followed and the corresponding FOI at the center of the cube should look like

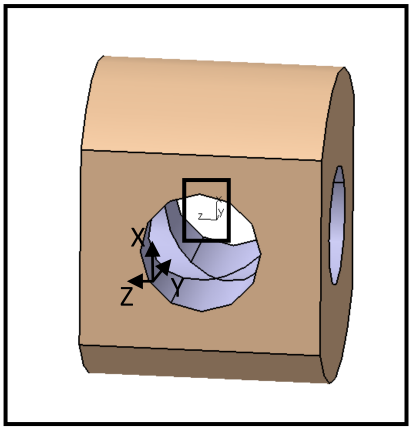
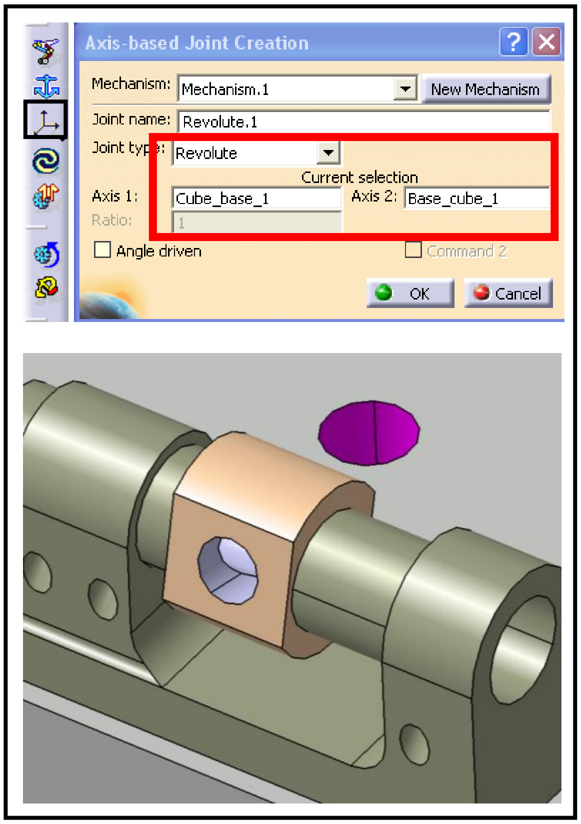


Figure B.10. Creating FOIs in the cube

- At this moment, the creation of joint will be done. First, click *Joint From Axis* button a new menu will appear and select *Revolute* as joint type and then for Axis 1 and 2 choose the FOIs created. The order for this work does not matter. Then click *OK* and a new revolute joint will be created and the cube part will be automatically placed at its corresponding point in the *Base* part. The menu for joint creation should look like



Axis-based Joint Creation

Mechanism: Mechanism.1

Joint name: Revolute.1

Joint type: Revolute

Current selection

Axis 1: Cube_base_1

Axis 2: Base_cube_1

Ratio: 1

□ Angle driven

□ Command 2

Figure B.11. Creating a revolute joint

- Now, repeat the same procedure for the remaining 5 cubes that should be connected to *Base* part. The result should be looking like

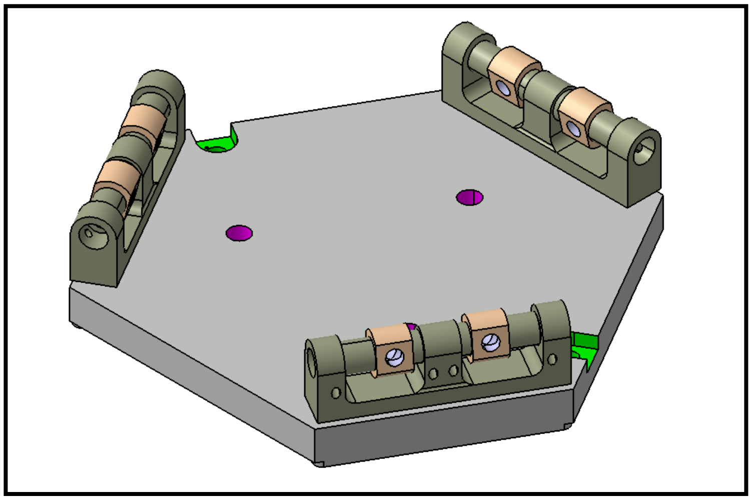


Figure B.12. Fixed part with 6 cubes

- At this point, the assembly of the lower legs to the cubes will be made. The FOIs will be appointed again. One for the lower cube with different Z-direction than the previous FOI. The other will be created on the lower leg. The respective FOIs should look like

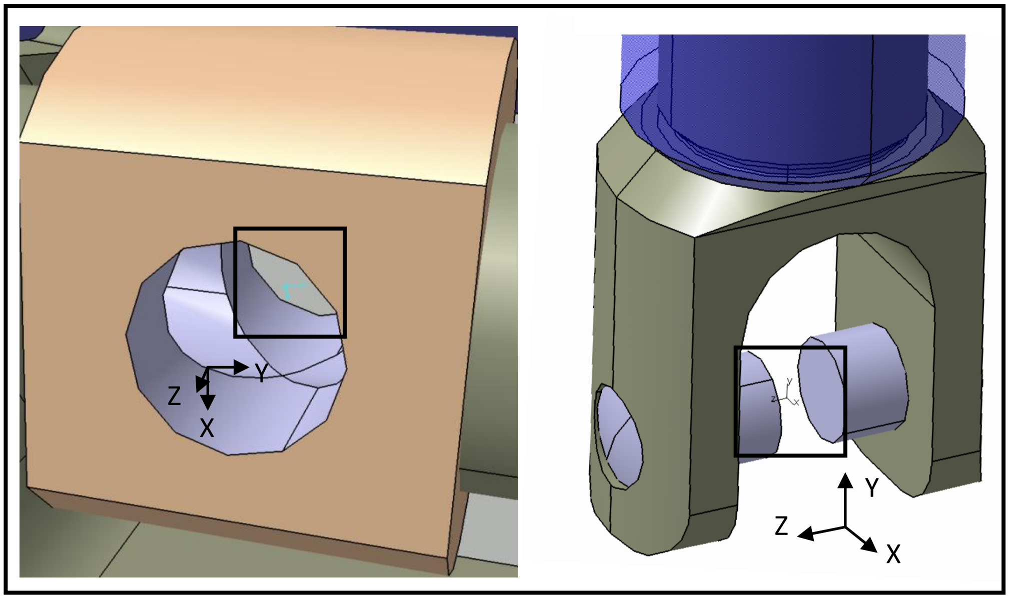


Figure B.13. FOIs for leg-assembly

- After the creation of FOIs, same procedure of joint creation for the cube will be followed. Then the result will be

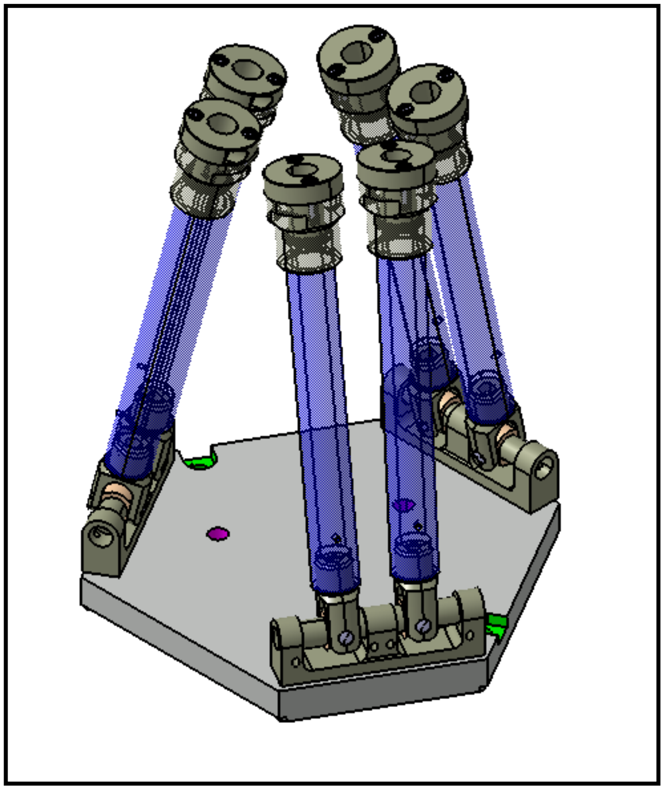


Figure B.14. Completed leg-assembly

- Now the creation of prismatic joints will be made. First, prospective FOIs will be created at joint zeros.
- The tricky point about prismatic joints that, when the joint is created DELMIA V5 takes exactly the positions of FOIs at that moment as zero point. It is because prismatic joints do not require FOIs to coincide at origins; thus, only coinciding in Z-direction any point can be zero point for the joint. This case is important to consider because when inverse kinematics is calculated, the joint range will be crucial and what is fed back as solution must be within that range to have a good posture. Therefore, it is very important to have a standard way of creating prismatic joints. One of these standard points will be presented here as well.
- The first FOI on *Lower Leg* part will be created the same way for previous parts. The respective FOI should be appointed at the zero point of the joint. It should look like

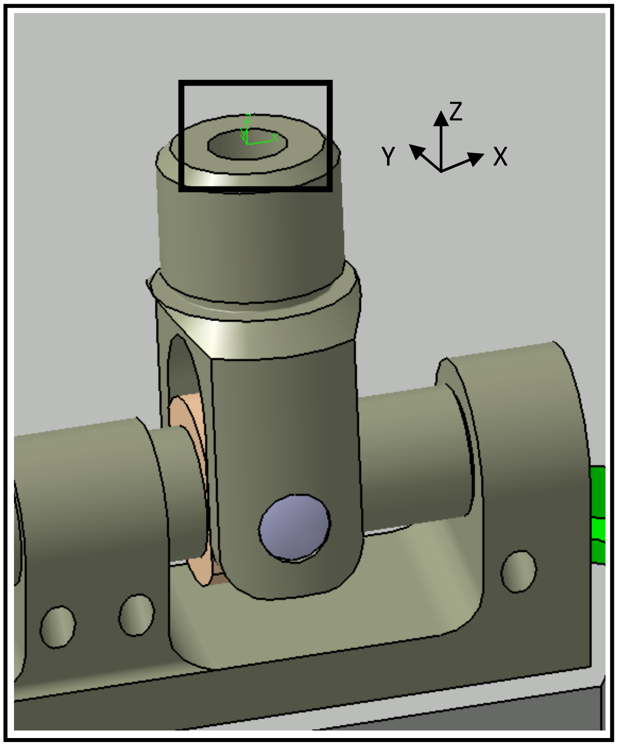


Figure B.15. FOI for the prismatic joint

- The second FOI will be created in the upper leg part at the bottom. The FOI will look like

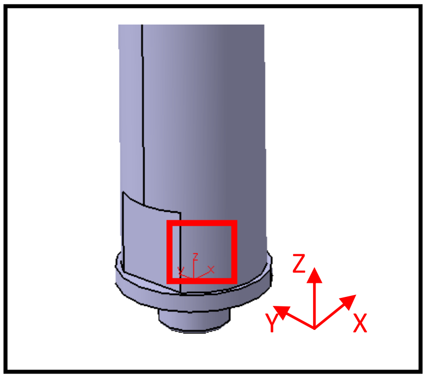
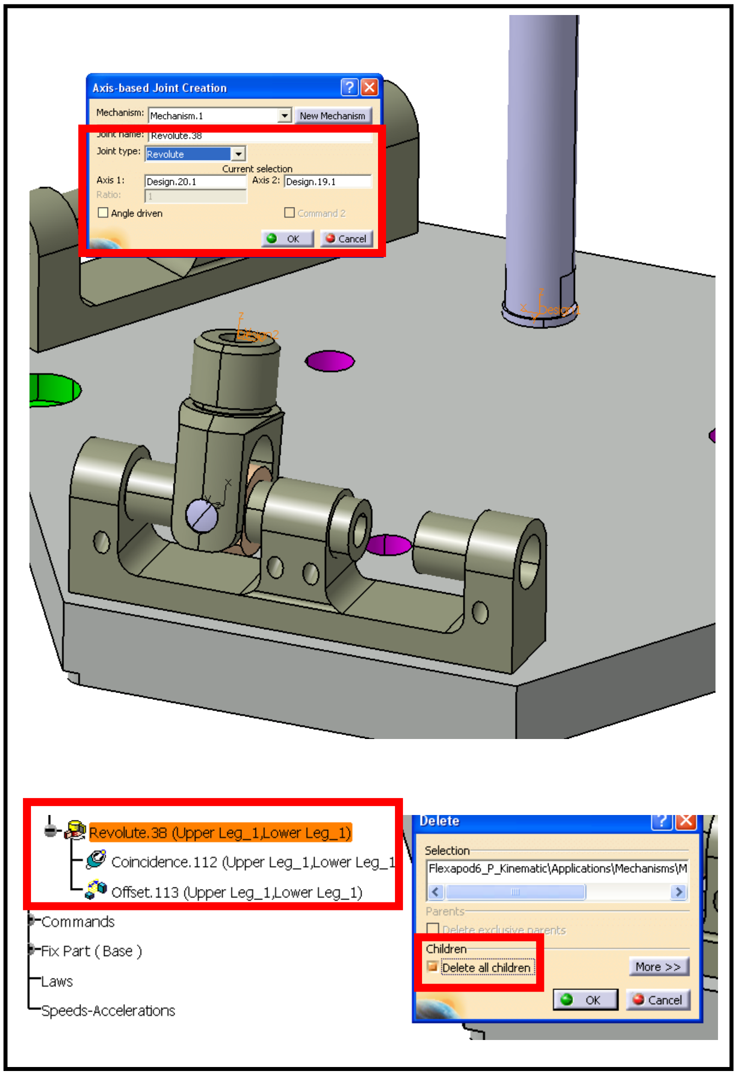


Figure B.16. The second FOI for the prismatic joint

- Normally, prismatic joints can be created immediately; but before, the origin points of FOIs must be coincided. The first way is to snap the *Upper Leg* part from the created FOI and propagate the part to the FOI of the *Lower Leg.* This can be done by copying-pasting the coordinates of the FOI of the *Lower Leg* to the FOI of *Upper Leg.*
- The second way is somewhat more error proof. The idea is to first create a revolute joint between the FOIs. This would result in a perfect coincidence at both Z-axes and origin points of each FOI. After the revolute joint creation, it must be deleted so that over that joint, a prismatic one can be created. One important point is to delete also the constraints that come with revolute joints. To make sure that there is no constraint left, delete the revolute joint with *Delete All Children* button activated. The steps than should look like



Mechanism: Mechanism.1

Joint name: Revolute.38

Joint type: Revolute

Axis 1: Design.20.1

Axis 2: Design.19.1

Ratio: 1

□ Angle driven

□ Command 2

Revolute.38 (Upper Leg_1,Lower Leg_1)

Coincidence.112 (Upper Leg_1,Lower Leg_1

Offset.113 (Upper Leg_1,Lower Leg_1)

Commands

Fix Part (Base)

Laws

Speeds-Accelerations

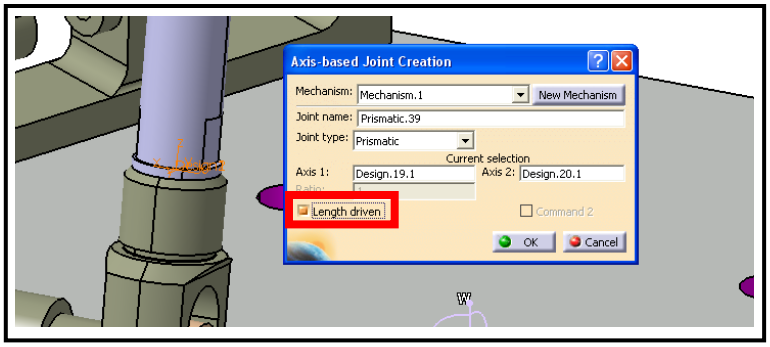
Flexapod6_P_Kinematic\Applications\Mechanisms\M

□ Delete exclusive parents

☑ Delete all children

Figure B.17. Revolute joint creation and deleting

- Note that when the revolute joint is deleted, the current positions of the parts remain the same. Without making any changes, prismatic joints will be created. Note also that in Flexapod 6P or hexapod, prismatic joints are actuators. Thus *Length Driven* is activated for this step. The steps in creation should look like


Axis-based Joint Creation

Mechanism: Mechanism.1

Joint name: Prismatic.39

Joint type: Prismatic

Current selection

Axis 1: Design.19.1

Axis 2: Design.20.1

Ratio: 1

☑ Length driven

□ Command 2

Figure B.18. Prismatic joint creation

- When these steps are repeated for other legs as well the result should be

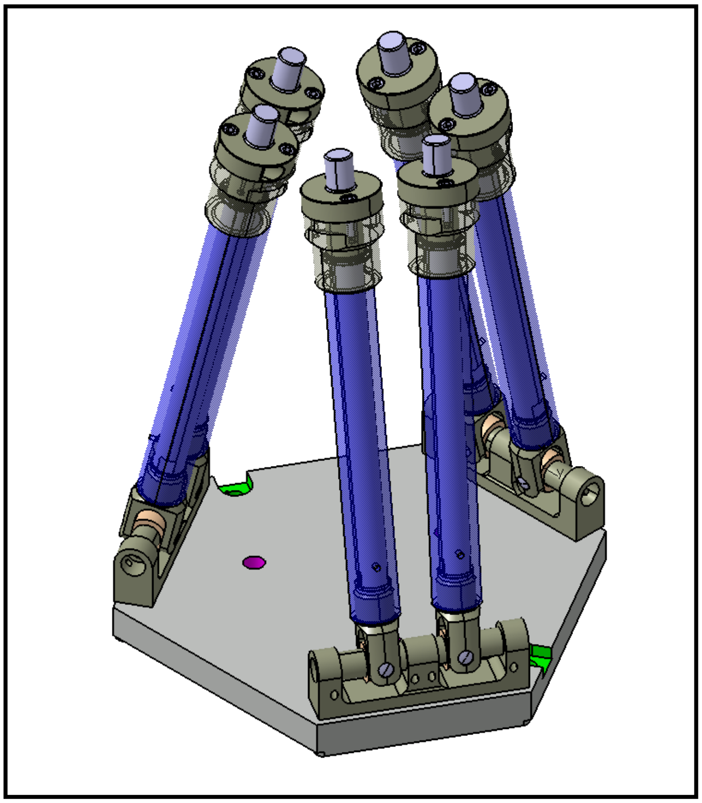


Figure B.19. Completed prismatic joint creation for all legs

- In this step, the intermediary part between *Upper Leg* and *Upper Cube* will be assembled. This part is named as Upper Leg Connection (ULC) that grants the third degree of freedom to the upper attachment point where the other two degrees of freedom are given by the *Upper Cube* part.
- As usual, first, FOIs will be created. The FOI for *ULC* part is

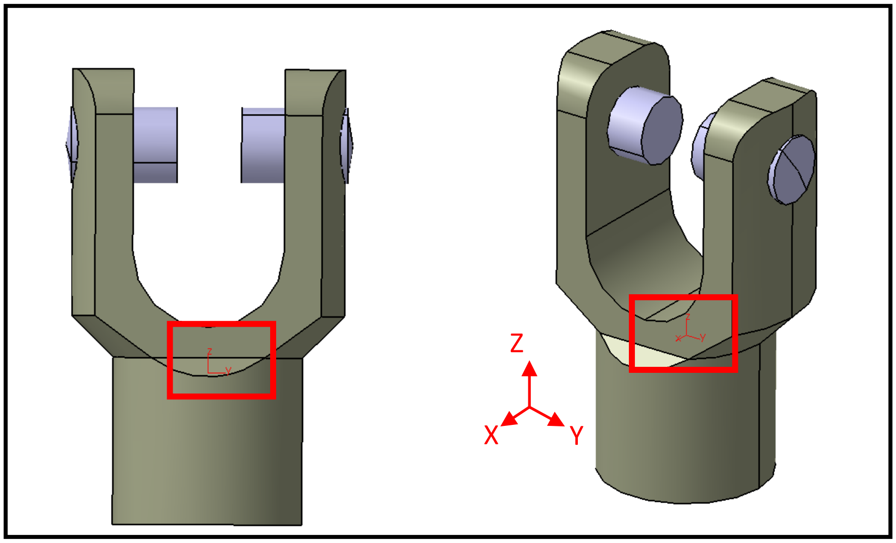


Figure B.20.FOIs for connecting parts

- The FOI for the *Upper Leg* part should be created at the top. Then the result should be

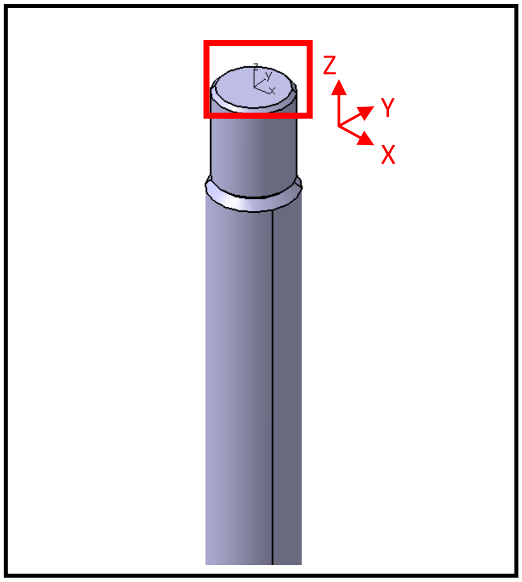
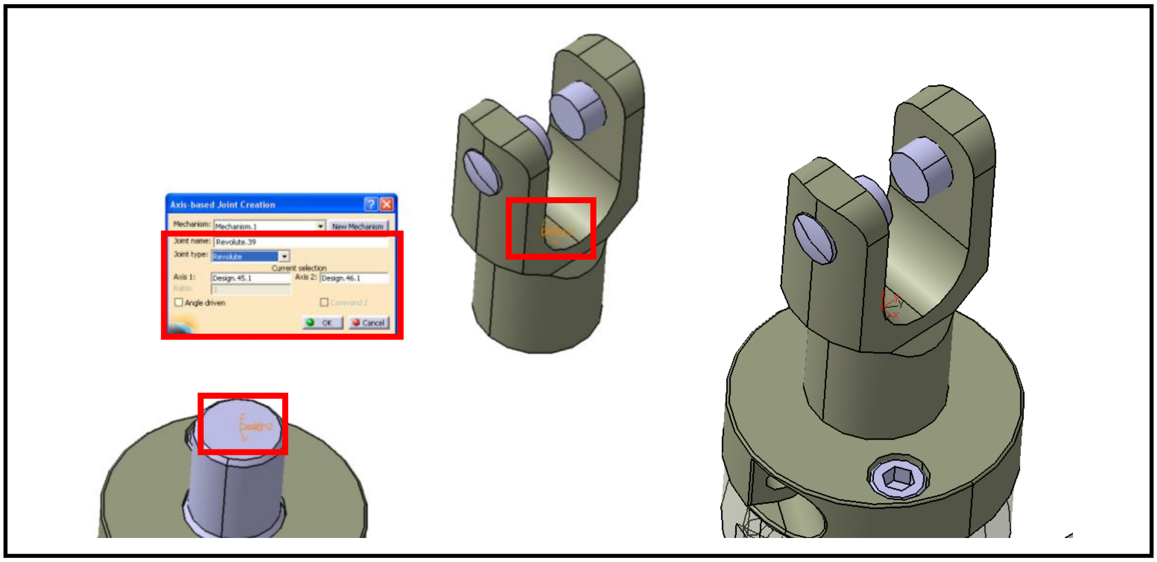


Figure B.21. Corresponding FOIs of connecting parts

- Now the revolute joint will be created.



Mechanism: Mechanism.1

Joint name: Revolute.39

Joint type: Revolute

Axis 1: Design.45.1

Axis 2: Design.46.1

Ratio: 1

□ Angle driven

□ Command 2

Figure B.22. Revolute joint creation for connecting parts

- Same operation will be repeated for the remaining 5 legs and the result will look like

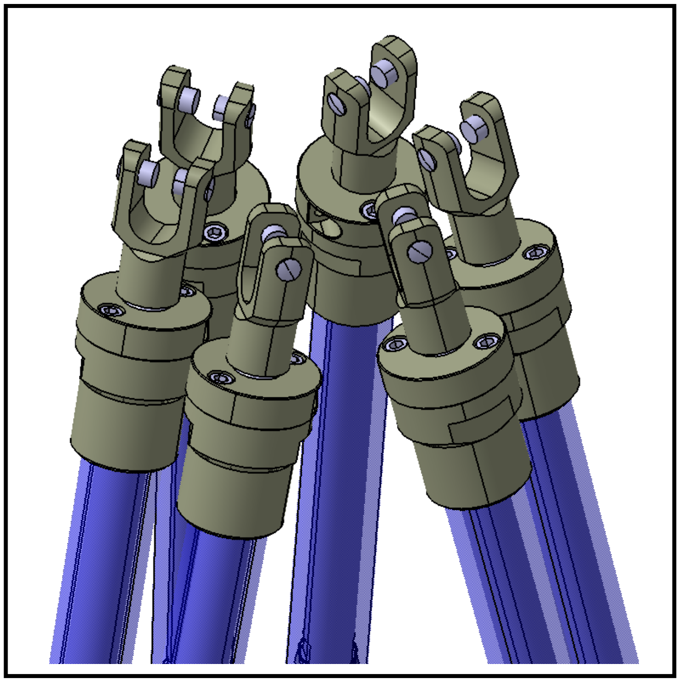


Figure B.23. Completed revolute joint creation

- In this step, the cubes that will connect the mobile platform to ULCs will be assembled. First, respective FOIs will be appointed to cubes and ULCs. The FOI of the cube will be the same as the lower cubes. The FOIs of the respective parts are then

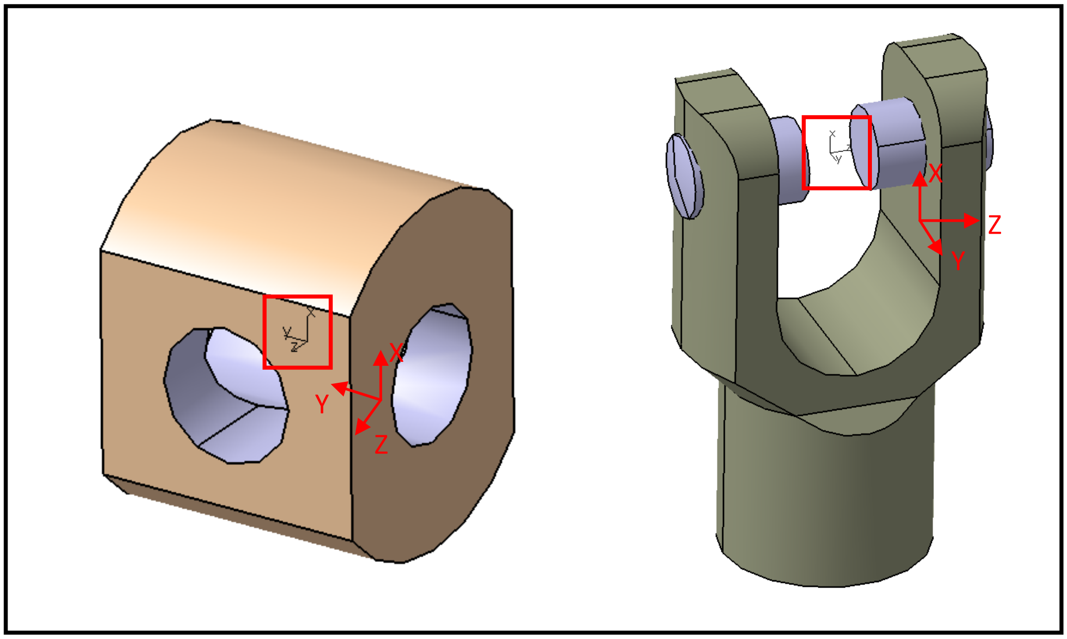
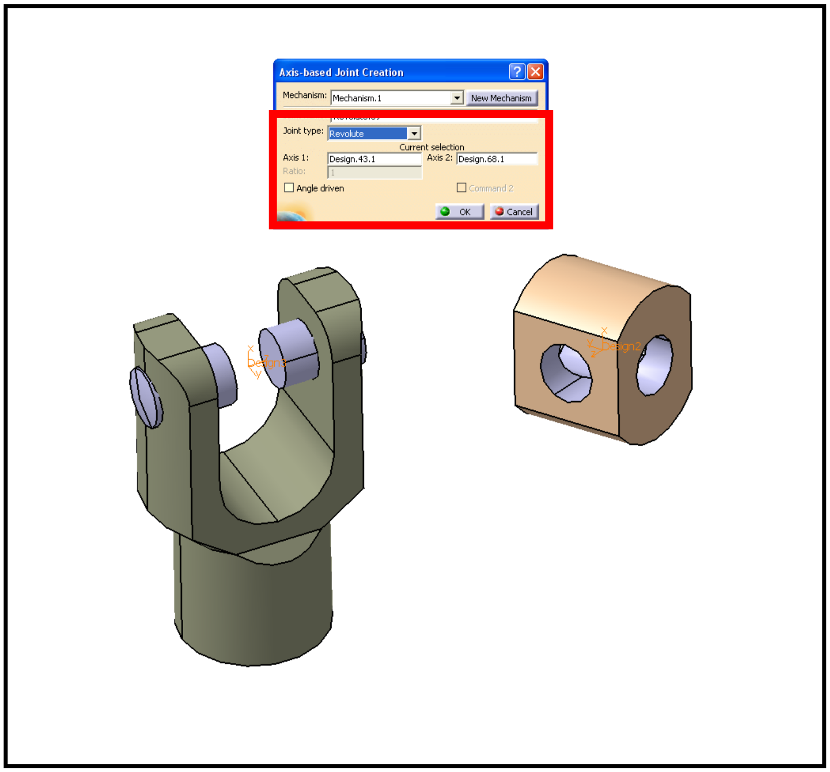


Figure B.24. FOIs for upper cubes and connecting parts

- Now the revolute joint will be created between these parts.


Axis-based Joint Creation

Mechanism: Mechanism.1 [New Mechanism]

Joint type: Revolute

Current selection

Axis 1: Design.43.1 Axis 2: Design.68.1

Ratio: 1

□ Angle driven □ Command 2

Figure B.25. Revolute joint creation

- When this operation is repeated for the remaining legs, the result will be

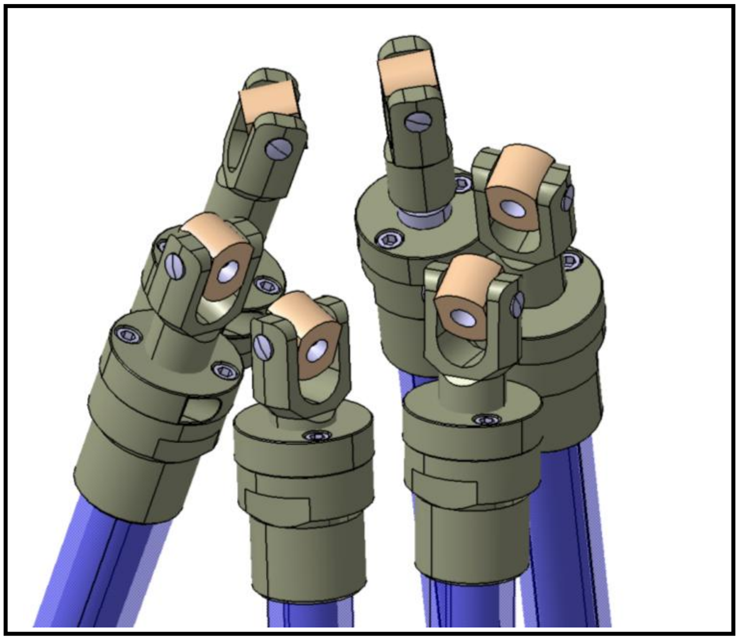


Figure B.26. Completed revolute joint creation

- The most important part of the assembly is to attach the mobile platform to the *Upper Cubes*. In order to achieve the right state, mobile platform must be fixed to some point with respect to the base platform. This way DELMIA will treat the legs as mobile and move the location of theirs in order to create the joints. If this way is not chosen, both legs and mobile platform will be moved and the result might be different than intended. An example of these unintended result

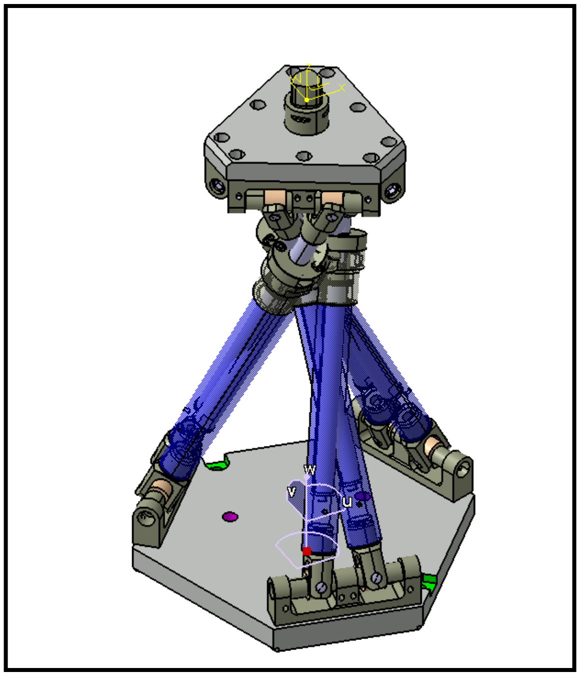


Figure B.27. Parts diving through each other

- In order to avoid this result or some other that is similar, the users must create a *Rigid Joint* between the base platform and the mobile. To achieve the right state, first the mobile platform should be propagated to the right coordinates. These coordinates are the ones that make all the joints zero and keep the mobile platform's Z-axis coincided with the base platform's Z-axis. The coordinates are $[0, 0, 500.515]^T$. The point of snapping is TCP and can be seen in the following figure

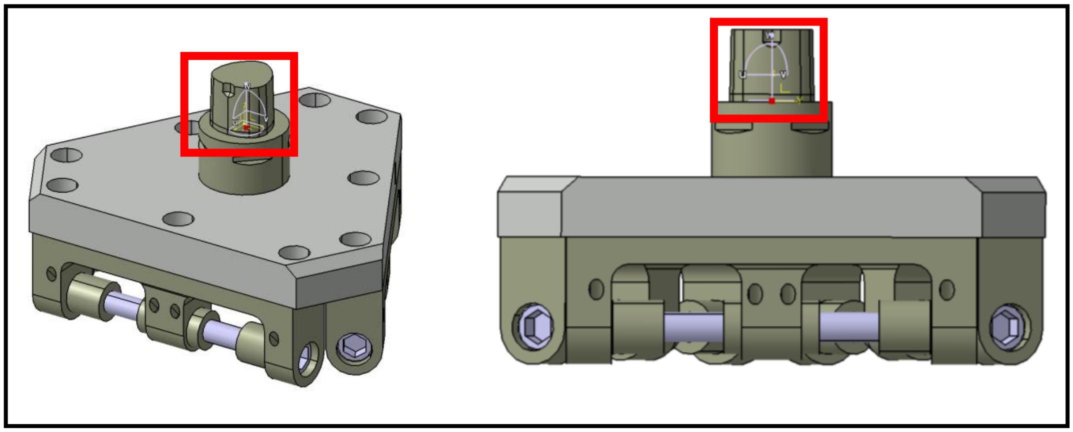
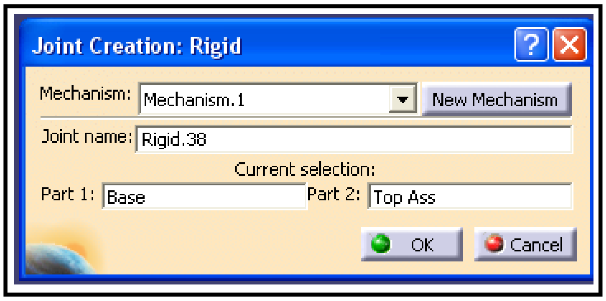


Figure B.28. Moving mobile platform

- The rigid joint then will be created by using *Rigid Joint* button. Choose the respective parts from the node tree and click *OK.* This way the mobile platform will be fixed to the base platform.



Joint Creation: Rigid

Mechanism: Mechanism.1

Joint name: Rigid.38

Current selection:

Part 1: Base

Part 2: Top Ass

Figure B.29. Creating a rigid joint

- After the creation of the rigid joint, it is now assured that the assembled legs will give the correct posture.
- In this step, the revolute joints between the *Upper Cubes* and the mobile platform. As standard, the respective FOIs will be produced first. The FOIs for the cubes and on the mobile platform will be at the center of the rotation. Thus, the FOIs will be

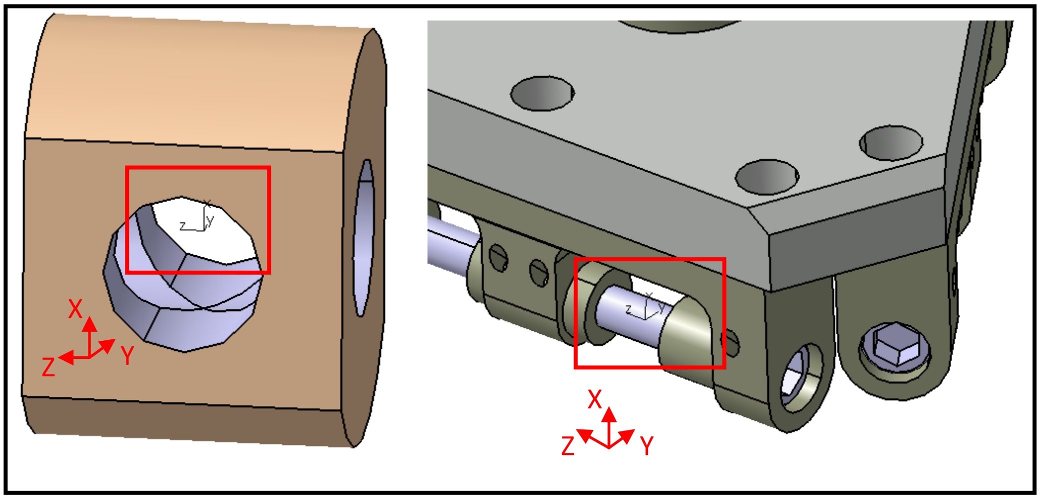
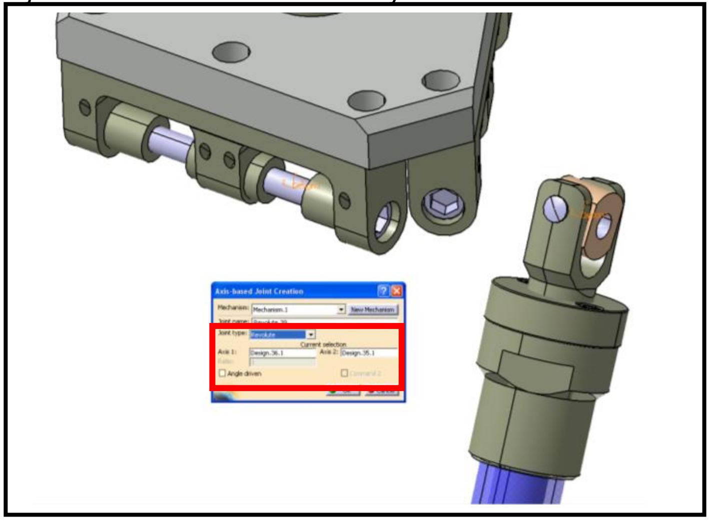


Figure B.30. Creating FOIs in the mobile platform and upper cubes

- Now revolute joint will be created with *Joint from axis* button.



Axis-based Joint Creation

Mechanism: Mechanism.1

New Mechanism

Joint name: Revolute.39

Joint type: Revolute

Current selection

Axis 1: Design.36.1

Axis 2: Design.35.1

Ratio: 1

□ Angle driven

□ Command 2

Figure B.31. Creating revolute joint

- Then, the result will be

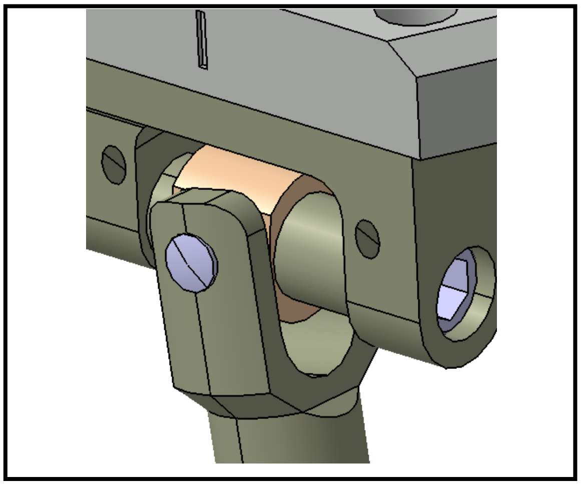


Figure B.32. Final revolute joint

- When this operation is repeated for the rest of the legs, the mechanism will be completed. The result of the top assembly will look like

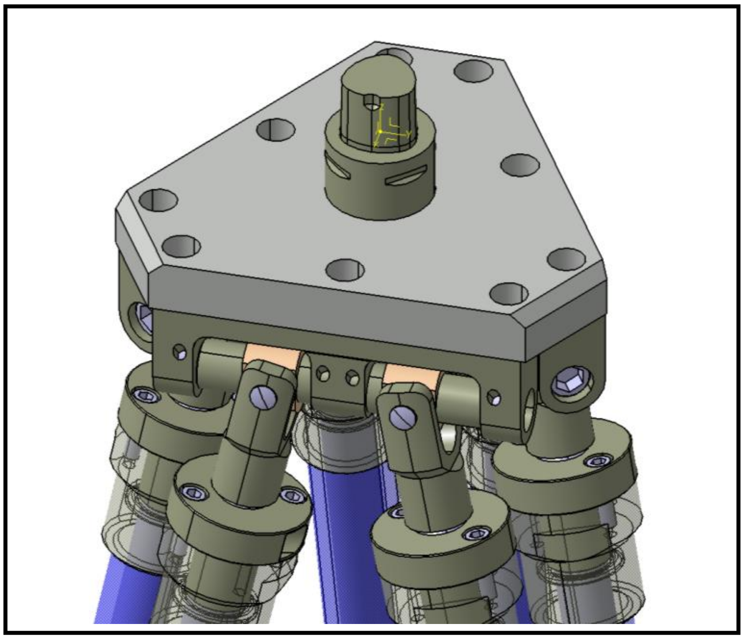


Figure B.33. Completed revolute joint creation

- The complete structure then should look like

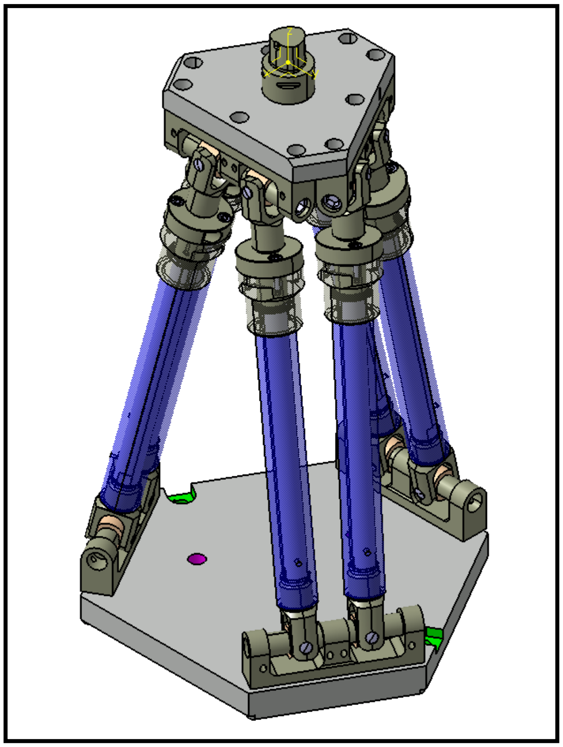


Figure B.34. Completed Flexapod 6P

- To complete the forward kinematics, users also have to define another FOI as TCP coordinate frame. The new FOI will be created under mobile platform in the same orientation as base coordinate frame since the starting transformation values should be standard position and same orientation as base. The FOI then should look like

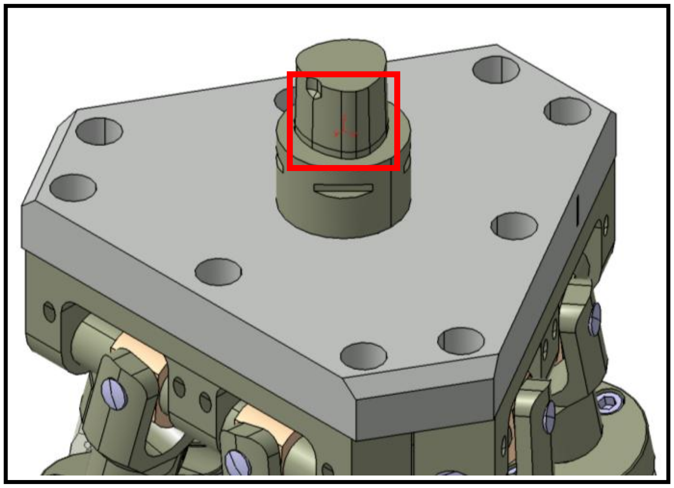
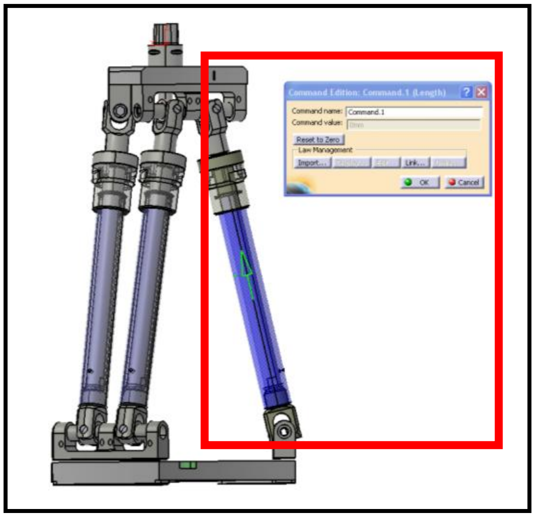


Figure B.35. FOI for TCP

- In this step, specifications regarding joints will be appointed. These specifications can be related to any joint, but in the case of Flexapod 6P these specifications are limited to actuators that are prismatic joints. The specifications are
    - Joint actuation direction
    - Joint range
    - Reference length
- The joint range can be arranged in two ways. The first way to define the limits for joints is double-clicking on the prismatic joint under *Mechanism-Joints* node. When clicked, *Joint Edition* menu will appear and on that menu users can define the travel limits and joint's actuation direction for each joint. The *Joint Edition* menu for the first prismatic joint will be



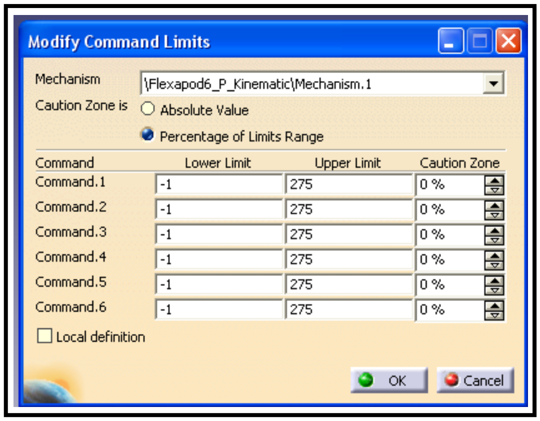
Command Edition: Command.1 (Length)

Command name: Command.1

Command value: 0mm

Figure B.36. Joint control

- The direction of joint is highlighted and the users can change the positive direction of the joint accordingly.
- The most convenient way to define limits is to click *Travel Limits* button and in the new menu define the joint limits for actuators. The travel limits for Flexapod 6P are [-1,275] mm. The reason that the lower limit is less than zero is the fact that when inverse kinematics is calculated, the code in the C-file might generate negative values that are very close to zero. This way a correct posture is guaranteed to be within the limits when inverse kinematics is loaded. Then, the menu for travel limits should look like



Modify Command Limits

Mechanism: \Flexapod6_P_Kinematic\Mechanism.1

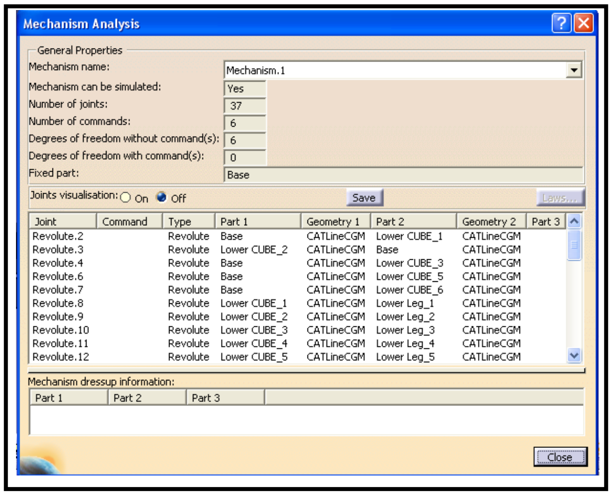
Caution Zone is ○ Absolute Value
● Percentage of Limits Range

Command	Lower Limit	Upper Limit	Caution Zone
Command.1	-1	275	0 %
Command.2	-1	275	0 %
Command.3	-1	275	0 %
Command.4	-1	275	0 %
Command.5	-1	275	0 %
Command.6	-1	275	0 %

□ Local definition

Figure B.37. Defining travel limits

- Second part of arranging properties of the joints can be done by clicking *Mechanism Properties* button. When clicked, a new menu will pop up, and on that menu users can see degrees of freedom with and without command, and also if the mechanism can be simulated. In addition, the user can see the joints and their constituting parts. The menu looks like



Mechanism Analysis

General Properties

Mechanism name: Mechanism.1
Mechanism can be simulated: Yes
Number of joints: 37
Number of commands: 6
Degrees of freedom without command(s): 6
Degrees of freedom with command(s): 0
Fixed part: Base

Joints visualisation: ○ On ● Off

Joint	Command	Type	Part 1	Geometry 1	Part 2	Geometry 2	Part 3
Revolute.2		Revolute	Base	CATLineCGM	Lower CUBE_1	CATLineCGM	
Revolute.3		Revolute	Lower CUBE_2	CATLineCGM	Base	CATLineCGM	
Revolute.4		Revolute	Base	CATLineCGM	Lower CUBE_3	CATLineCGM	
Revolute.6		Revolute	Base	CATLineCGM	Lower CUBE_5	CATLineCGM	
Revolute.7		Revolute	Base	CATLineCGM	Lower CUBE_6	CATLineCGM	
Revolute.8		Revolute	Lower CUBE_1	CATLineCGM	Lower Leg_1	CATLineCGM	
Revolute.9		Revolute	Lower CUBE_2	CATLineCGM	Lower Leg_2	CATLineCGM	
Revolute.10		Revolute	Lower CUBE_3	CATLineCGM	Lower Leg_3	CATLineCGM	
Revolute.11		Revolute	Lower CUBE_4	CATLineCGM	Lower Leg_4	CATLineCGM	
Revolute.12		Revolute	Lower CUBE_5	CATLineCGM	Lower Leg_5	CATLineCGM	

Mechanism dressup information:

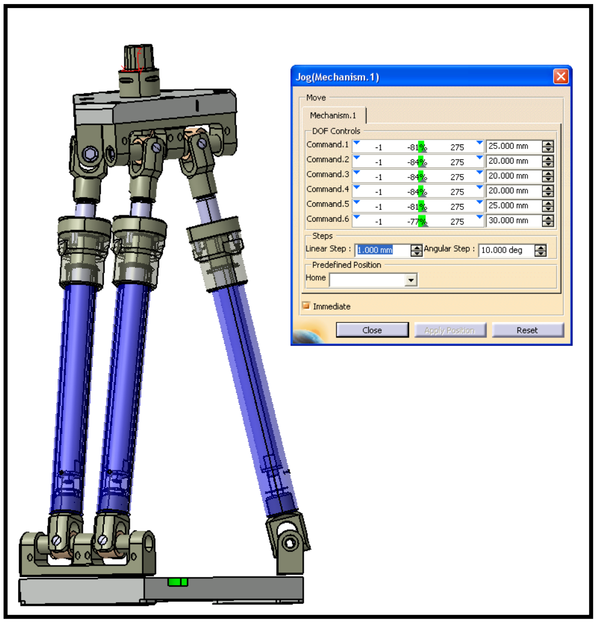
Part 1	Part 2	Part 3

Figure B.38. Mechanism analysis

- When the definition and arrangements are made, the testing of the created mechanism will be made.

TESTING OF THE FLEXAPOD 6P'S FORWARD KINEMATICS

- The testing of the forward kinematics is simply done by testing the behavior of the joints for different values on actuators. This is done by *Jog Mechanism* button. When clicked, a new menu will appear and on that menu users can manipulate the actuators. If the joints are responding as intended then it means the built mechanism is working properly.



Jog(Mechanism.1)

Move — Mechanism.1 — DOF Controls

Command.1	-1	-81%	275	25.000 mm
Command.2	-1	-84%	275	20.000 mm
Command.3	-1	-84%	275	20.000 mm
Command.4	-1	-84%	275	20.000 mm
Command.5	-1	-81%	275	25.000 mm
Command.6	-1	-77%	275	30.000 mm

Steps

Linear Step : 1.000 mm Angular Step : 10.000 deg

Predefined Position

Home

Immediate

Figure B.39. Mechanism testing

- If an error is observed, it is possible to go to joints menu and check their properties once again.

THE CREATION OF INVERSE KINEMATICS

- Before creating the inverse kinematics, the users have to create a part that will be attached to the mobile platform's TCP. Thus, go to *Insert-New part* and create a part under the parent node. Then, name the part accordingly. In this case, the new part is named as *TCP.*

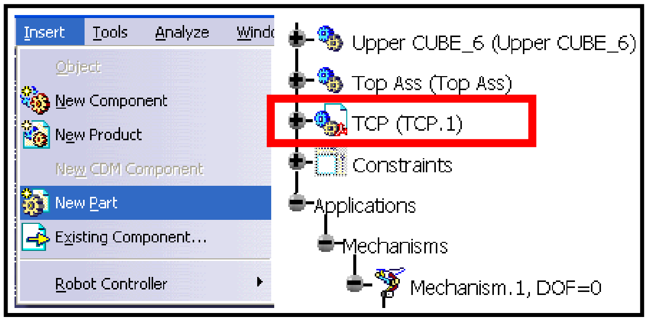
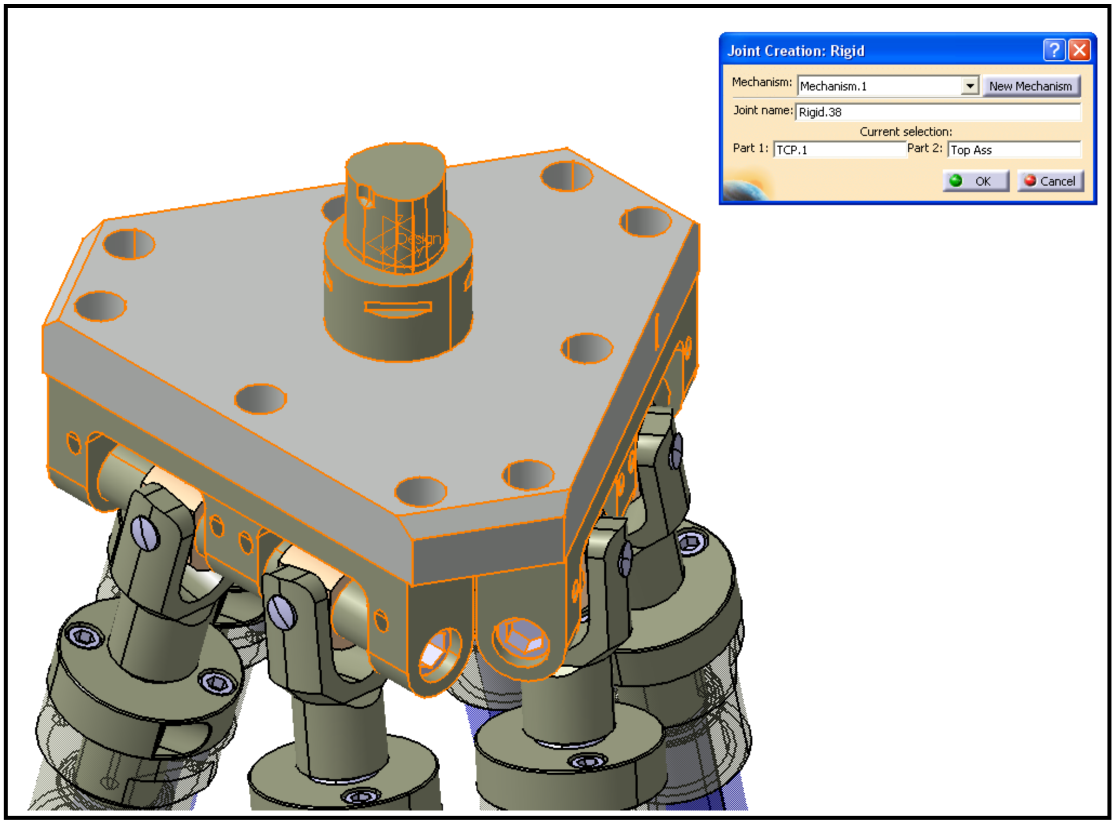


Figure B.40. Creating a new part for TCP

- After the creation of the new part, propagate it to the same location to the TCP. To achieve it, take the coordinates of FOI located at TCP and apply it to the new part. After the propagation, create a rigid joint between the new part and the mobile platform.



Joint Creation: Rigid

Mechanism: Mechanism.1

Joint name: Rigid.38

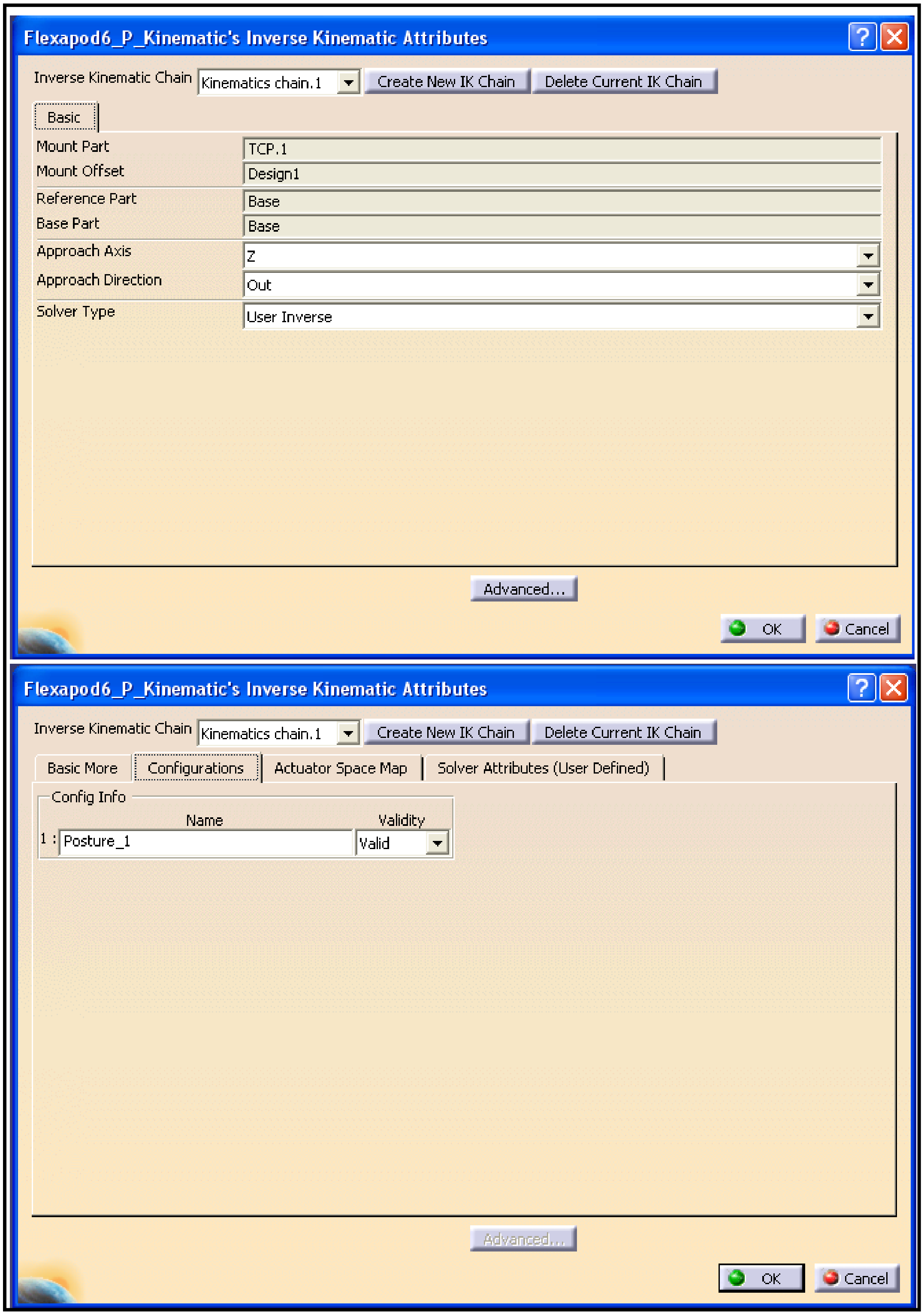
Current selection:

Part 1: TCP.1 Part 2: Top Ass

Figure B.41. Attaching TCP to mobile platform

- When the new rigid joint is created, the inverse kinematics is now ready to be defined.
- The inverse kinematics is created through the button *Inverse Kinematics* . Click the button and select the parent element on the node tree. A new menu will appear. On the menu, appoint accordingly as below
    - **Mount Part:** TCP.1 (the part created after the mechanism creation named as TCP)
    - **Mount Offset:** The FOI created to represent TCP
    - **Reference part:** Base platform
    - **Base:** Base platform
    - **Approach axis:** Z-axis
    - **Solver type:** User Inverse (for C-file method)
- After appointing the sections in *Basic* tab, click *Advanced*. Note that new tabs now appear next to *Basic* tab.

- In the *Configurations* tab, make sure that the *Posture_1* is valid. Then proceed to the next tab *Actuator Space Map*.



Flexapod6_P_Kinematic's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

Mount Part	TCP.1
Mount Offset	Design1
Reference Part	Base
Base Part	Base
Approach Axis	Z
Approach Direction	Out
Solver Type	User Inverse

Flexapod6_P_Kinematic's Inverse Kinematic Attributes

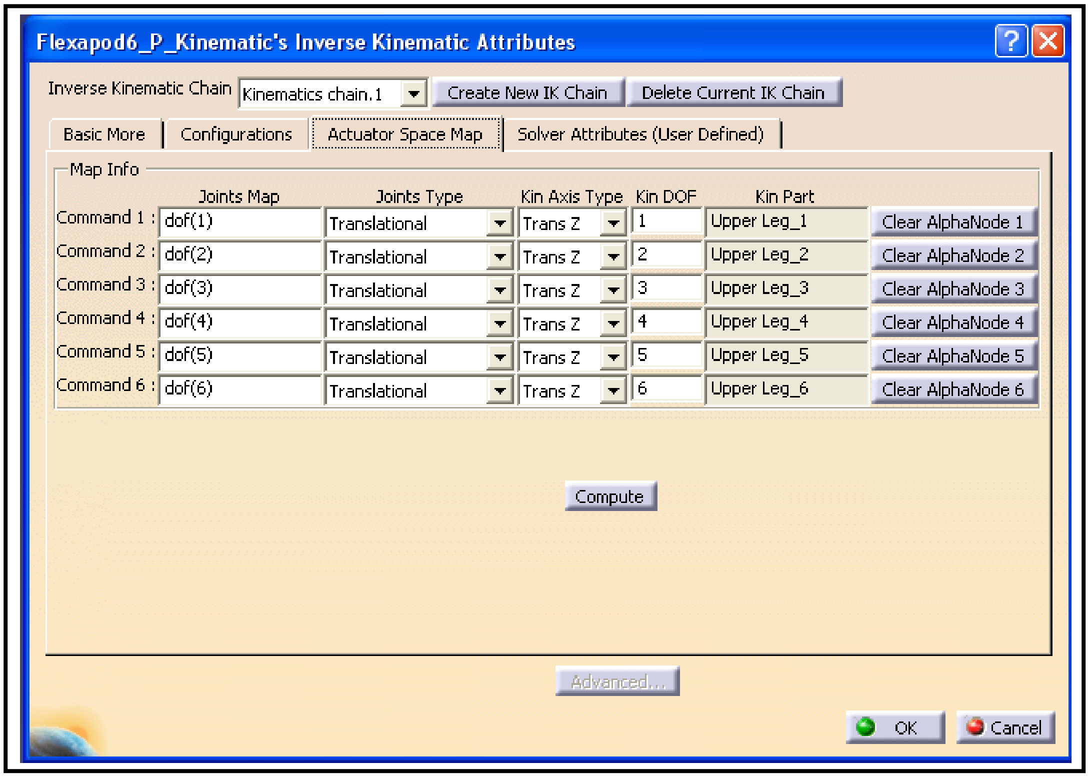
Inverse Kinematic Chain: Kinematics chain.1

Config Info

	Name	Validity
1 :	Posture_1	Valid

Figure B.42. Defining Inverse Kinematics parameters of DELMIA V5

- In the *Actuator Space Map* tab, appoint each leg to the corresponding *dof*. These *dof*s and Kin DOF parameters are
  - **Leg 1 (Command 1) :** dof(1) – Translational –Trans Z- Kin DOF 1
  - **Leg 2 (Command 2) :** dof(2) – Translational –Trans Z- Kin DOF 2
  - **Leg 3 (Command 3) :** dof(3) – Translational –Trans Z- Kin DOF 3
  - **Leg 4 (Command 4) :** dof(4) – Translational –Trans Z- Kin DOF 4
  - **Leg 5 (Command 5) :** dof(5) – Translational –Trans Z- Kin DOF 5
  - **Leg 6 (Command 1) :** dof(6) – Translational –Trans Z- Kin DOF 6
- The *Kin Part* section asks the user to appoint the mobile parts of the prismatic joints. In this case these parts are *Upper Legs.* To appoint the corresponding parts, click on the *Kin Part* area and then go to node tree and choose the corresponding part. These parts are
  - **Leg 1 (Command 1) :** Upper_Leg_1
  - **Leg 2 (Command 2) :** Upper_Leg_2
  - **Leg 3 (Command 3) :** Upper_Leg_3
  - **Leg 4 (Command 4) :** Upper_Leg_4
  - **Leg 5 (Command 5) :** Upper_Leg_5
  - **Leg 6 (Command 6) :** Upper_Leg_6
- The corresponding figure can be seen below for this tab.



Flexapod6_P_Kinematic's Inverse Kinematic Attributes

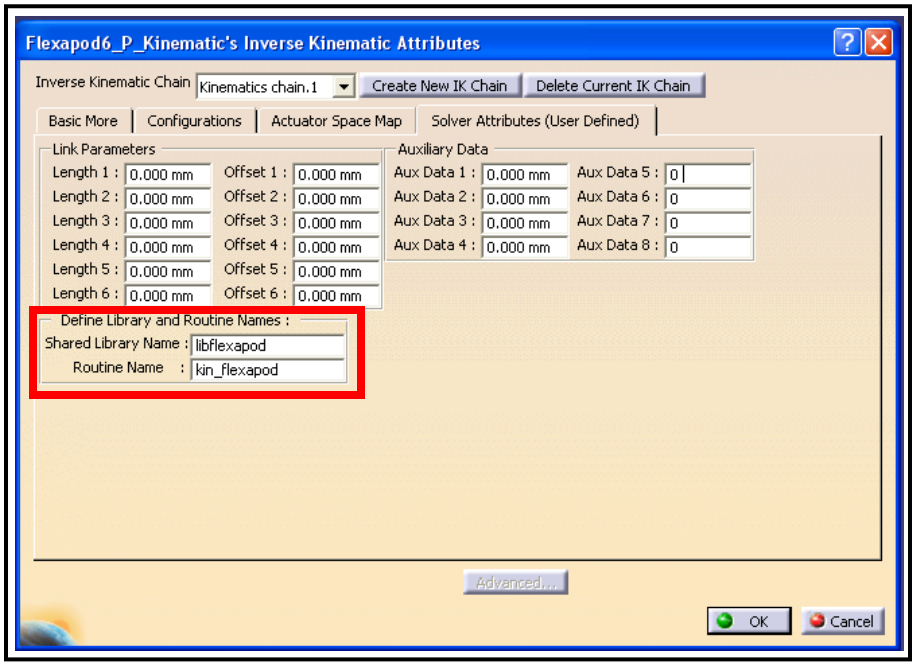
Inverse Kinematic Chain: Kinematics chain.1

Map Info

	Joints Map	Joints Type	Kin Axis Type	Kin DOF	Kin Part	
Command 1 :	dof(1)	Translational	Trans Z	1	Upper Leg_1	Clear AlphaNode 1
Command 2 :	dof(2)	Translational	Trans Z	2	Upper Leg_2	Clear AlphaNode 2
Command 3 :	dof(3)	Translational	Trans Z	3	Upper Leg_3	Clear AlphaNode 3
Command 4 :	dof(4)	Translational	Trans Z	4	Upper Leg_4	Clear AlphaNode 4
Command 5 :	dof(5)	Translational	Trans Z	5	Upper Leg_5	Clear AlphaNode 5
Command 6 :	dof(6)	Translational	Trans Z	6	Upper Leg_6	Clear AlphaNode 6

Figure B.43. Defining Actuator's parameters

- The last tab and operation is *Solver Attributes* tab. In this tab, only the names of the C-code and library files will be entered. The remaining sections should be left empty since the code covers these values. Hence, the tab should look like



Flexapod6_P_Kinematic's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

Link Parameters

Length 1 :	0.000 mm	Offset 1 :	0.000 mm
Length 2 :	0.000 mm	Offset 2 :	0.000 mm
Length 3 :	0.000 mm	Offset 3 :	0.000 mm
Length 4 :	0.000 mm	Offset 4 :	0.000 mm
Length 5 :	0.000 mm	Offset 5 :	0.000 mm
Length 6 :	0.000 mm	Offset 6 :	0.000 mm

Auxiliary Data

Aux Data 1 :	0.000 mm	Aux Data 5 :	0
Aux Data 2 :	0.000 mm	Aux Data 6 :	0
Aux Data 3 :	0.000 mm	Aux Data 7 :	0
Aux Data 4 :	0.000 mm	Aux Data 8 :	0

Define Library and Routine Names :

Shared Library Name : libflexapod

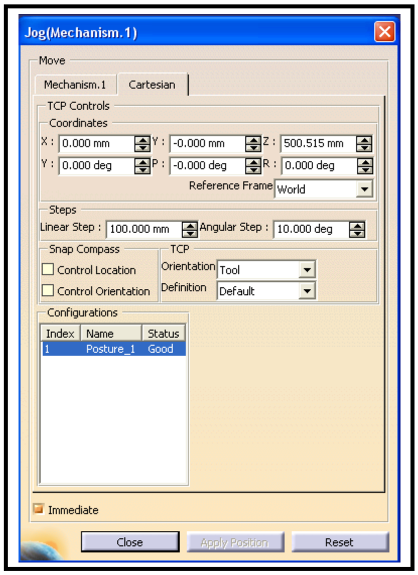
Routine Name : kin_flexapod

Figure B.44. Defining library and C-file names

- Then click OK, and the inverse kinematics definition will be complete.

TESTING OF THE INVERSE MECHANISM

- First click on *Jog Mechanism* button and see that there is a new tab in the menu called *Cartesian*. In this menu, DELMIA allows its users to manipulate the mechanism by using TCP tag – which is only activated when the inverse kinematics is defined. The menu then



Jog(Mechanism.1)

X : 0.000 mm Y : -0.000 mm Z : 500.515 mm

Y : 0.000 deg P : -0.000 deg R : 0.000 deg

Reference Frame World

Linear Step : 100.000 mm Angular Step : 10.000 deg

□ Control Location

□ Control Orientation

Orientation Tool

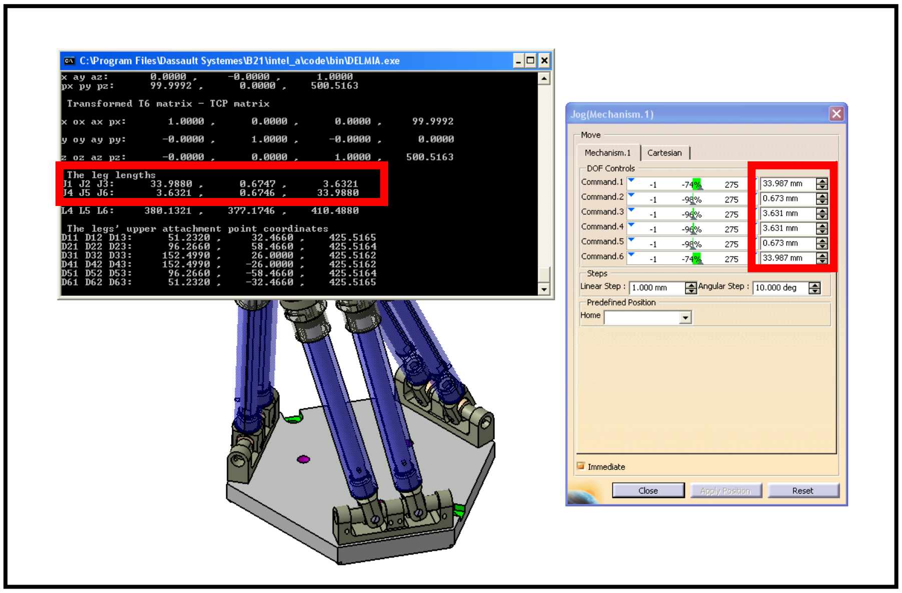
Definition Default

Index	Name	Status
1	Posture_1	Good

☑ Immediate

Figure B.45. Jogging mechanism with inverse kinematics definition

- To test the mechanism, one may enter the TCP coordinates and orientation or simply drag the TCP tag to some random locations.
- When the movement is made, go to *Mechanism.1* tab and check the joint values with results printed on the debugging window. If they match each other, then the inverse kinematics is working as intended.



```
x ay az:       0.0000 ,     -0.0000 ,      1.0000
px py pz:      99.9992 ,     0.0000 ,    500.5163

 Transformed T6 matrix - TCP matrix

x ox ax px:       1.0000 ,      0.0000 ,      0.0000 ,     99.9992

y oy ay py:      -0.0000 ,      1.0000 ,     -0.0000 ,      0.0000

z oz az pz:      -0.0000 ,      0.0000 ,      1.0000 ,    500.5163

 The leg lengths
J1 J2 J3:      33.9880 ,      0.6747 ,      3.6321
J4 J5 J6:       3.6321 ,      0.6746 ,     33.9880

L4 L5 L6:     380.1321 ,    377.1746 ,    410.4880

 The legs' upper attachment point coordinates
D11 D12 D13:      51.2320 ,     32.4660 ,    425.5165
D21 D22 D23:      96.2660 ,     58.4660 ,    425.5164
D31 D32 D33:     152.4990 ,     26.0000 ,    425.5162
D41 D42 D43:     152.4990 ,    -26.0000 ,    425.5162
D51 D52 D53:      96.2660 ,    -58.4660 ,    425.5164
D61 D62 D63:      51.2320 ,    -32.4660 ,    425.5165
```

Command.1 -1 -74% 275 33.987 mm
Command.2 -1 -98% 275 0.673 mm
Command.3 -1 -96% 275 3.631 mm
Command.4 -1 -96% 275 3.631 mm
Command.5 -1 -98% 275 0.673 mm
Command.6 -1 -74% 275 33.987 mm

Linear Step : 1.000 mm Angular Step : 10.000 deg

Home

Figure B.46. Comparing results to debugging window

THE C-FILE FOR INVERSE KINEMATICS OF FLEXAPOD 6P

In section 4.2, the relevant theory was transferred to a C-file for hexapod. The notations and the way of working are also described in that section. Thus, here the complete code will be given.

```
/****************************************************************************
***
**
**                        USER KINEMATICS EXAMPLE
**
**   Copyright (c) 1990 Delmia Corporation, All rights reserved.
**
**   This file contains an example of a kinematics routine for the
**   shared library.  This example will work for 4 DOF 2 Config (left and
right
**   elbow) scara robots such as the ASEA/IRB300.  By default,
**   kin_usr1 is mapped to this routine.
**
**   For a description of kinematics solutions refer to:
**
**       Paul, Richard P., "Robot Manipulators: Mathematics, Programming
**       and Control", The MIT Press, Cambridge, Massachusetts, 1981.
**
**   DESCRIPTION OF ARGUMENTS
**
**   double T6[4][4]        4x4 position matrix of center of wrist. This is
**                         the goal point MINUS the tool frame and mounting
**                         plate offsets. This is the easiest point to start
**                         the inverse kinematic solution from, and is the
**                         traditional approach.
**
**                         NOTE: T6 matrix may be transposed from your usual
**                         notation.
**
**                              | nx ny nz 0 | \\
**                         T6 = | ox oy oz 0 |  > direction cosines (9)
**                              | ax ay az 0 | /
**                              | px py pz 1 | -> position terms (3)
**
**                         px = T6[3][0];
**
**  double link_lengths[]  Distance between joint axis along link length
**
**  double link_offsets[]  Offset between joint axis along joint axis
**
**                         These two arrays can be considered the Denevitt-
**                         Hartenburg variables described in Paul's book, or
**                         any convenient scheme the user desires.
**
**  double solutions[][]   A two dimensional array contains all possible
**                         solutions for robot arm. It is up to user to
**                         decide how many solutions are possible, and to
**                         provide all solutions when routine is called:
**                         elbow up, elbow down, etc.  The CONFIGS
**                         Button in IGRIP allows user to view all possible
**                         solutions and may provide insight into importance
**                         of this array.
**
**  int warnings[]         Array providing warning states for each solution
**                         such as unreachable, singular, etc. Possible
warning
**                         states are defined in include file shlibdefs.h
**                         and are:
**
**                         WARN_GOOD_SOLUTION
**                         WARN_JOINT_LIMIT_EXCEEDED
**                         WARN_UNREACHABLE
**                         WARN_SINGULAR_SOLUTION
**
**       NOTE: shlibdefs.h is automatically included by the IGRIP Shared
**             Library Make system.  For further details regarding the
building
**             of the shared library, refer to the IGRIP Motion Pipeline
**             Reference Guide
**
**
**  Words of encouragement
**
**      Writing inverse kinematics routines is a challenge. Invariably
**      you will make mistakes which later seem trivial.  Even experts on
**      the subject loathe writing a new routine.  The usual problems
**      are matching the routines view of the world with the device
**      definition.  You must check that where this routine thinks is
**      the axis origin, or the zero reference position, is the same
**      as the IGRIP device.  Also make sure that each agree upon the
positive
**      sense of direction.  These are the most common foul ups.  Next,
**      the mounting plate offset may be wrong, so when first debugging
**      your routine, set the mounting plate and tool frame offsets to
**      zero.  Next check for dropped signs in your equations.  Maybe
**      an inverse trig function is returning an angle in a different
quadrant
**      than the one you want.  Perhaps you should be using atan2 instead
**      of atan (or vice-versa).  Remember that trig and inverse trig
function
**      angles are in radians.  Also, check array indices.  Remember that
**      arrays start at zero not one, so link_4's offset is at
link_offsets[3].
**      Are you referring to T6[3][2], when you mean T6[2][3]? Remember that
**      transformation matrices may be transposed from standard text book
**      definitions.  Once you get your routine to work you will have earned
**      the title of kinematician.
**
*****************************************************************************
**/

#include <shlibdefs.h>

/*
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
**
** USER SHOULD CHANGE THESE VALUES APPROPRIATELY
**                       |
**                       |
**                      \ /
**                       v                                         */

#define NUM_SOLUTIONS    1        /* Number of possible solutions  */
#define NUM_DOFS         6        /* Number of joints to be solved */

/*                       ^
**                      / \
**                       |
**                       |
** USER SHOULD CHANGE THESE VALUES APPROPRIATELY
**
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
```

```
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
*/


/*
 * User must supply this function
*/

DllExport int
get_kin_config( char *kin_routine, int *kin_dof, int *solution_count, int
*usrKinDataHint )
{
      if( strcmp( kin_routine, "kin_flexapod" ) == 0 )
      {
            *kin_dof = NUM_DOFS;
            *solution_count = NUM_SOLUTIONS;
                  /*
                  * this indicates kin_usr's last argument (void *pData)
                  * will be DLM_Data_KinStat
                  */
                  *usrKinDataHint = USR_KIN_DATA_KINSTAT;
                  return 0;
      }
      return 1;
}

static char JointType[2][24] = { "ROTATIONAL", "TRANSLATIONAL" };
static char KinMode[2][24] = { "Normal", "TrackTCP" };

/*
** Routine Name
*/
DllExport int
kin_flexapod(
      link_lengths,
      link_offsets,
      T6, /* See above for description of these arguments */
      solutions,
      warnings,
            pData
      )
/*
** Passed Variable Declarations
*/
double T6[4][4],
      link_lengths[],
      link_offsets[],
      solutions[][NUM_SOLUTIONS];
int warnings[];


void *pData; /* usr routine should NEVER delete pData */

{
```

```
/*
** Local Variable Declarations (add variable declarations as appropriate)
*/
long double nx, ny, nz, ox, oy, oz, ax, ay, az, px, py, pz;
long double D11, D12, D13, D21, D22, D23, D31, D32, D33, D41, D42, D43, D51,
D52, D53, D61, D62, D63;
long double L1,L2,L3,L4,L5,L6,J1,J2,J3,J4,J5,J6, Lref;

//Variables to perform matrix multiplication
int row1,row2,row3,row4,row5,row6;
int col1,col2,col3,col4,col5,col6;
int inner1,inner2,inner3,inner4,inner5,inner6;

// The upper attachmentpoints for each leg (The vector between the TCP and
each upper attachment point).
long double L1tToTCP[4][1];
long double L2tToTCP[4][1];
long double L3tToTCP[4][1];
long double L4tToTCP[4][1];
long double L5tToTCP[4][1];
long double L6tToTCP[4][1];

//The transformed T6 matrix named as TCP matrix (do not confuse with Tool
Centre Point) - see line 307.
long double TCP[4][4];

//The current position for each upper attachmentpoint (in Base coordinates).
long double L1tCur[4] = {0};
long double L2tCur[4] = {0};
long double L3tCur[4] = {0};
long double L4tCur[4] = {0};
long double L5tCur[4] = {0};
long double L6tCur[4] = {0};

//Lower attachemnt points on each leg (in Base coordinates)
long double L1b[3]; long double L2b[3]; long double L3b[3];
long double L4b[3]; long double L5b[3]; long double L6b[3];

#if 1
/*
* using pData
*/
      int i;

      DLM_Data_KinStat *pDLM_Data = (DLM_Data_KinStat *) pData;
      if( pDLM_Data )
      {
            printf( "\n\ndof_count: %d\n", pDLM_Data->dof_count );

            printf( "\njoint_types:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%s ", JointType[(pDLM_Data->joint_types)[i]] );

            printf( "\n\nkin_mode: %s\n", KinMode[pDLM_Data->kin_mode] );

            printf( "\njoint_values:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
```

```
                  printf( "%12.4f ", pDLM_Data->joint_values[i] );

            printf( "\n\njnt_trvl_lmts lower:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[0][i] );

            printf( "\n\njnt_trvl_lmts upper:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[1][i] );

            printf( "\n\n" );

      }

#endif
/***--------------- Execution Begins Here ----------------------------------
***/
      /*
      ** DO NOT REMOVE THIS BLOCK OF CODE
      ** IT IS REQUIRED TO PROPERLY SET THE NUMBER OF KINEMATIC
      ** DOFS FOR THE DEVICE
      */
      if( !kin_check_definition( NUM_DOFS, NUM_SOLUTIONS ) )
      {
      /*
      ** Inconsistency between device definition and inverse
      ** kinematics routine exists. A warning message has been
      ** issued and routine aborted
      */
      return( 1 );
      }

/***---------------- User code begins here ---------------------------------
***/



//The vector between the TCP and the upper attachmentpoints for each leg
L1tToTCP[0][0] = -48.767; L1tToTCP[1][0] =  32.466; L1tToTCP[2][0] = -75;
L1tToTCP[3][0] = 1;

L2tToTCP[0][0] =  -3.733; L2tToTCP[1][0] =  58.466; L2tToTCP[2][0] = -75;
L2tToTCP[3][0] = 1;

L3tToTCP[0][0] =    52.5; L3tToTCP[1][0] =      26; L3tToTCP[2][0] = -75;
L3tToTCP[3][0] = 1;

L4tToTCP[0][0] =    52.5; L4tToTCP[1][0] =     -26; L4tToTCP[2][0] = -75;
L4tToTCP[3][0] = 1;

L5tToTCP[0][0] =  -3.733; L5tToTCP[1][0] = -58.466; L5tToTCP[2][0] = -75;
L5tToTCP[3][0] = 1;

L6tToTCP[0][0] = -48.767; L6tToTCP[1][0] = -32.466; L6tToTCP[2][0] = -75;
L6tToTCP[3][0] = 1;
```

```
//The lower attachmentpoints for each leg (in Base-coordinates).
L1b[0]      = -132.5;   L1b[1] =       26;      L1b[2] = 58.5;
L2b[0]      = 43.733;   L2b[1] =  127.748;      L2b[2] = 58.5;
L3b[0]      = 88.767;   L3b[1] =  101.748;      L3b[2] = 58.5;
L4b[0]      = 88.767;   L4b[1] = -101.748;      L4b[2] = 58.5;
L5b[0]      = 43.733;   L5b[1] = -127.748;      L5b[2] = 58.5;
L6b[0]      = -132.5;   L6b[1] =      -26;      L6b[2] = 58.5;
//Importing the current TCP values from Delmia through the T6 matrix and
putting proper context
nx = T6[0][0];
ny = T6[0][1];
nz = T6[0][2];
ox = T6[1][0];
oy = T6[1][1];
oz = T6[1][2];
ax = T6[2][0];
ay = T6[2][1];
az = T6[2][2];
px = T6[3][0];
py = T6[3][1];
pz = T6[3][2];
//Printing the current TCP values in the debug window for evaluation purposes
printf( "\nx ny nz: %12.4f ,%12.4f ,%12.4f\n", nx ,ny ,nz );
printf( "\ox oy oz: %12.4f ,%12.4f ,%12.4f\n", ox ,oy ,oz );
printf( "\ax ay az: %12.4f ,%12.4f ,%12.4f\n", ax ,ay ,az );
printf( "\px py pz: %12.4f ,%12.4f ,%12.4f\n", px ,py ,pz );
//The transforming T6 matrix from row vectors form to column vector form
TCP[0][0] =  nx; TCP[0][1] =  ox; TCP[0][2] =  ax; TCP[0][3] = px;
TCP[1][0] =  ny; TCP[1][1] =  oy; TCP[1][2] =  ay; TCP[1][3] = py;
TCP[2][0] =  nz; TCP[2][1] =  oz; TCP[2][2] =  az; TCP[2][3] = pz;
TCP[3][0] =  0;  TCP[3][1] = 0;   TCP[3][2] =  0;  TCP[3][3] = 1;
//Printing the transformed matrix TCP
printf( "\n Transformed T6 matrix - TCP matrix\n");
printf( "\nx ox ax px: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", nx ,ox ,ax, px );
printf( "\ny oy ay py: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", ny ,oy ,ay, py );
printf( "\nz oz az pz: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", nz ,oz ,az, pz );


//Calculating the current position (in x,y,z in Base coordinates) of each
upper attachment point for each leg by multiplying the transformation
// matrix TCP[4][4] with the vector between the current TCP (the T6 matrix)
and the upper attachmentpoint for each leg (LxToTCP[][])

//Calculate upper position on Leg1 (The array L1tCur)
 for (row1 = 0; row1 < 4; row1++) {
        for (col1 = 0; col1 < 1; col1++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner1 = 0; inner1 < 4; inner1++) {
                L1tCur[row1] += TCP[row1][inner1] * L1tToTCP[inner1][col1];
            }
        }
  }
```

```
//Calculate upper position on Leg2 (The array L2tCur)
    for (row2 = 0; row2 < 4; row2++) {
        for (col2 = 0; col2 < 1; col2++) {


             // Multiply the row of A by the column of B to get the row,
      column of product.
            for (inner2 = 0; inner2 < 4; inner2++) {
                L2tCur[row2] += TCP[row2][inner2] * L2tToTCP[inner2][col2];
            }
        }
  }
//Calculate upper position on Leg3 (The array L3tCur)
    for (row3 = 0; row3 < 4; row3++) {
        for (col3 = 0; col3 < 1; col3++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner3 = 0; inner3 < 4; inner3++) {
                L3tCur[row3] += TCP[row3][inner3] * L3tToTCP[inner3][col3];
            }
        }
  }
//Calculate upper position on Leg4 (The array L4tCur)
    for (row4 = 0; row4 < 4; row4++) {
        for (col4 = 0; col4 < 1; col4++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner4 = 0; inner4 < 4; inner4++) {
                L4tCur[row4] += TCP[row4][inner4] * L4tToTCP[inner4][col4];
            }
        }
  }
//Calculate upper position on Leg5 (The array L5tCur)
    for (row5 = 0; row5 < 4; row5++) {
        for (col5 = 0; col5 < 1; col5++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner5 = 0; inner5 < 4; inner5++) {
                L5tCur[row5] += TCP[row5][inner5] * L5tToTCP[inner5][col5];
            }
        }
  }

//Calculate upper position on Leg6 (The array L6tCur)
    for (row6 = 0; row6 < 4; row6++) {
        for (col6 = 0; col6 < 1; col6++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner6 = 0; inner6 < 4; inner6++) {
                L6tCur[row6] += TCP[row6][inner6] * L6tToTCP[inner6][col6];
            }
        }
  }
```

```
// Calcultates the distance between the upper and lower attachment points for
each leg.
      L1 = sqrt(((pow((L1tCur[0]-L1b[0]),2)))+((pow((L1tCur[1]-
L1b[1]),2)))+((pow((L1tCur[2]-L1b[2]),2))));
      L2 = sqrt(((pow((L2tCur[0]-L2b[0]),2)))+((pow((L2tCur[1]-
L2b[1]),2)))+((pow((L2tCur[2]-L2b[2]),2))));
      L3 = sqrt(((pow((L3tCur[0]-L3b[0]),2)))+((pow((L3tCur[1]-
L3b[1]),2)))+((pow((L3tCur[2]-L3b[2]),2))));
      L4 = sqrt(((pow((L4tCur[0]-L4b[0]),2)))+((pow((L4tCur[1]-
L4b[1]),2)))+((pow((L4tCur[2]-L4b[2]),2))));
      L5 = sqrt(((pow((L5tCur[0]-L5b[0]),2)))+((pow((L5tCur[1]-
L5b[1]),2)))+((pow((L5tCur[2]-L5b[2]),2))));
      L6 = sqrt(((pow((L6tCur[0]-L6b[0]),2)))+((pow((L6tCur[1]-
L6b[1]),2)))+((pow((L6tCur[2]-L6b[2]),2))));

//The distance between upper and lower leg attachmentpoint when the command
joint is zero. Used as a reference to get the current leg length
Lref = 376.5;

//Calculates the joint values by calulating the differnce in distance between
the two attachmentpoints on each leg and a reference length (Lref)
//(the lenght between attachment points when the joints are 0)
J1 =  L1 - Lref;
J2 =  L2 - Lref;
J3 =  L3 - Lref;
J4 =  L4 - Lref;
J5 =  L5 - Lref;
J6 =  L6 - Lref;

//Sending the final joint values back to the "solutions"-matrix which is the
input matrix for Delmia.
solutions[0][0] = J1;
solutions[1][0] = J2;
solutions[2][0] = J3;
solutions[3][0] = J4;
solutions[4][0] = J5;
solutions[5][0] = J6;

//Printing some of the variable values out in the debug window to ease
debugging and get an overview of what is going on
printf( "\n The leg lengths\n" );
printf( "J1 J2 J3: %12.4f ,%12.4f ,%12.4f\n", J1 ,J2 ,J3 );
printf( "J4 J5 J6: %12.4f ,%12.4f ,%12.4f\n", J4 ,J5 ,J6 );
printf( "L1 L2 L3: %12.4f ,%12.4f ,%12.4f\n", L1 ,L2 ,L3 );
printf( "L4 L5 L6: %12.4f ,%12.4f ,%12.4f\n", L4 ,L5 ,L6);

D11 = L1tCur[0]; D12 = L1tCur[1]; D13 = L1tCur[2];
D21 = L2tCur[0]; D22 = L2tCur[1]; D23 = L2tCur[2];
D31 = L3tCur[0]; D32 = L3tCur[1]; D33 = L3tCur[2];
D41 = L4tCur[0]; D42 = L4tCur[1]; D43 = L4tCur[2];
D51 = L5tCur[0]; D52 = L5tCur[1]; D53 = L5tCur[2];
D61 = L6tCur[0]; D62 = L6tCur[1]; D63 = L6tCur[2];
```

```
printf( "\n The legs' upper attachment point coordinates \n" );
printf( "\D11 D12 D13: %12.4f ,%12.4f ,%12.4f\n", D11 ,D12 ,D13 );
printf( "\D21 D22 D23: %12.4f ,%12.4f ,%12.4f\n", D21 ,D22 ,D23 );
printf( "\D31 D32 D33: %12.4f ,%12.4f ,%12.4f\n", D31 ,D32 ,D33 );
printf( "\D41 D42 D43: %12.4f ,%12.4f ,%12.4f\n", D41 ,D42 ,D43 );
printf( "\D51 D52 D53: %12.4f ,%12.4f ,%12.4f\n", D51 ,D52 ,D53 );
printf( "\D61 D62 D63: %12.4f ,%12.4f ,%12.4f\n", D61 ,D62 ,D63 );

warnings[ 0 ] = WARN_GOOD_SOLUTION;

return (0);

}
```

APPENDIX C: THE CREATION OF INVERSE KINEMATICS FOR HEXAPOD

The hexapod as stated in section 2.1 is an equivalent structure to Flexapod 6P where the upper attachment points are spherical joints instead of three successive revolute joints as with Flexapod 6P.

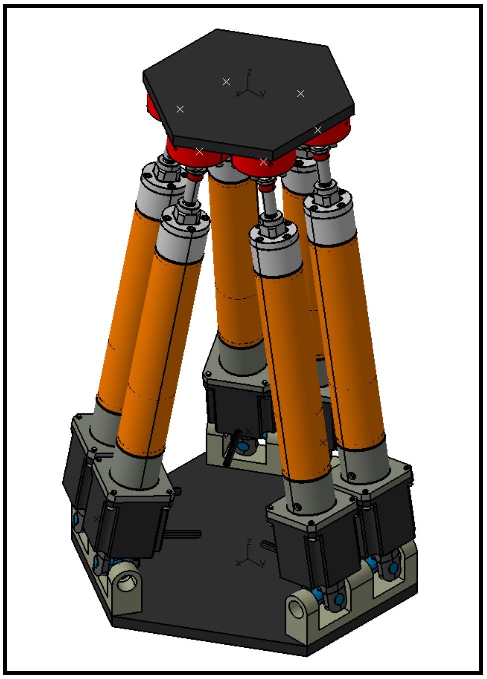
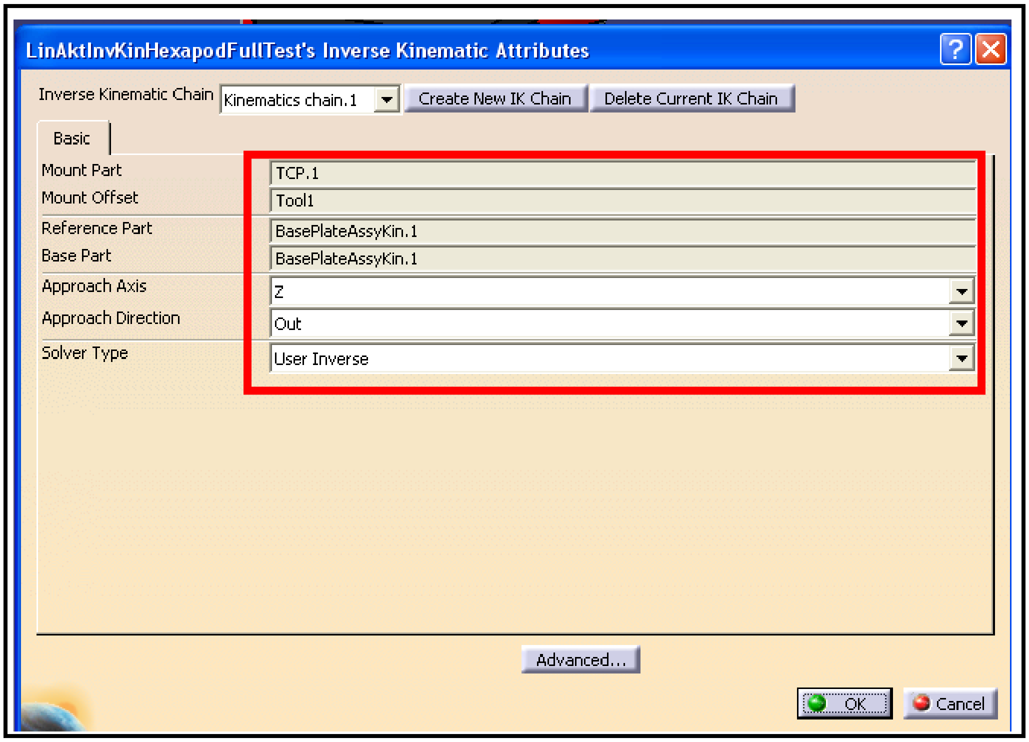


Figure C.1. Hexapod with forward kinematics created

As put earlier, the forward kinematics was received ready from the previous project owner thus in this appendix, the creation of forward kinematics will not be given. Hence, the inverse kinematics will be the topic of this section.

THE INVERSE KINEMATICS CREATION

- As with the Flexapod 6P case, the same procedure will be followed. Thus the *Basic* tab should look like
    - **TCP.1:** The part created to mount with the TCP point
    - **Tool1:** The FOI at TCP point
    - **BasePlateAssyKin.1 :** The base platform part
    - **Z:** Approach axis
    - **Out** is the approach direction
    - **User inverse method** is for C-file use

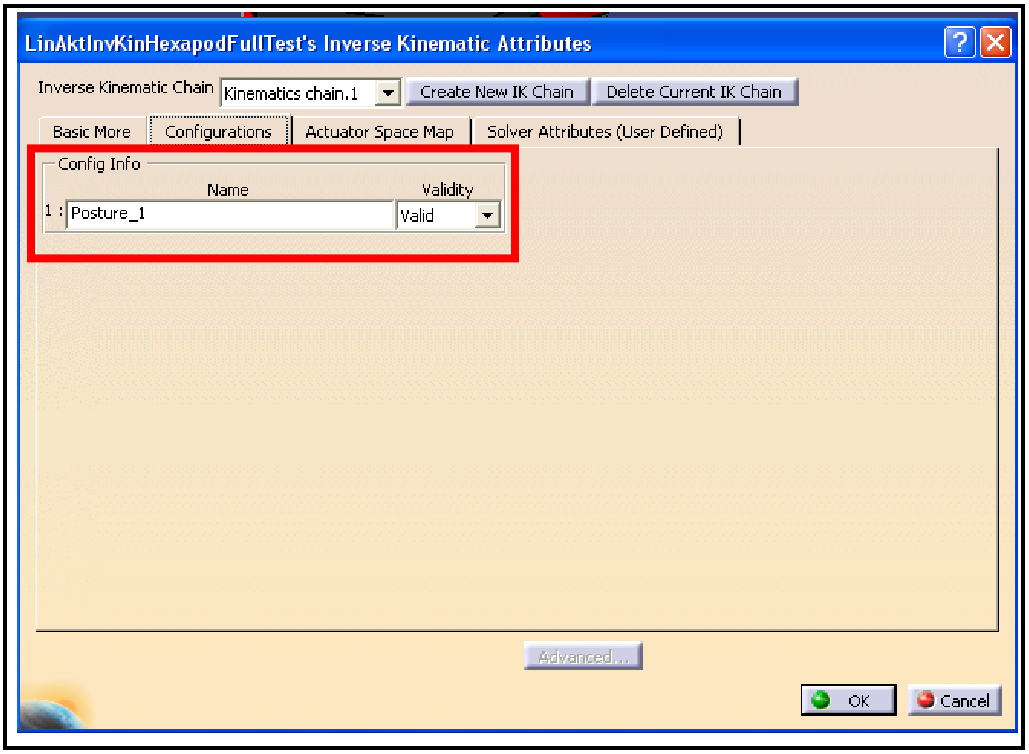

LinAktInvKinHexapodFullTest's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

Mount Part: TCP.1
Mount Offset: Tool1
Reference Part: BasePlateAssyKin.1
Base Part: BasePlateAssyKin.1
Approach Axis: Z
Approach Direction: Out
Solver Type: User Inverse

Figure C.2. Defining Inverse kinematics parameters

- When clicked *Advanced,* the remaining tabs are the same as Flexapod 6P. These tabs than


LinAktInvKinHexapodFullTest's Inverse Kinematic Attributes

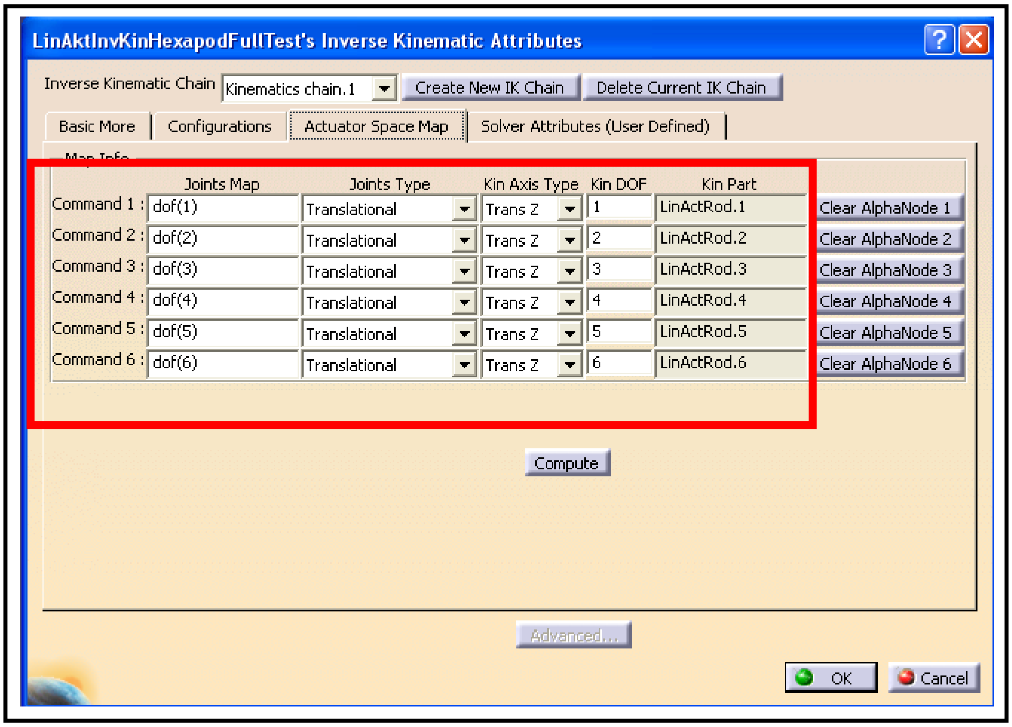
Inverse Kinematic Chain: Kinematics chain.1

Config Info

	Name	Validity
1 :	Posture_1	Valid

Figure C.3. Checking configurations

- The *Actuator Space Map* tab follows the same procedure with Flexapod 6P. The *Kin Part* has the mobile parts of the prismatic joints.

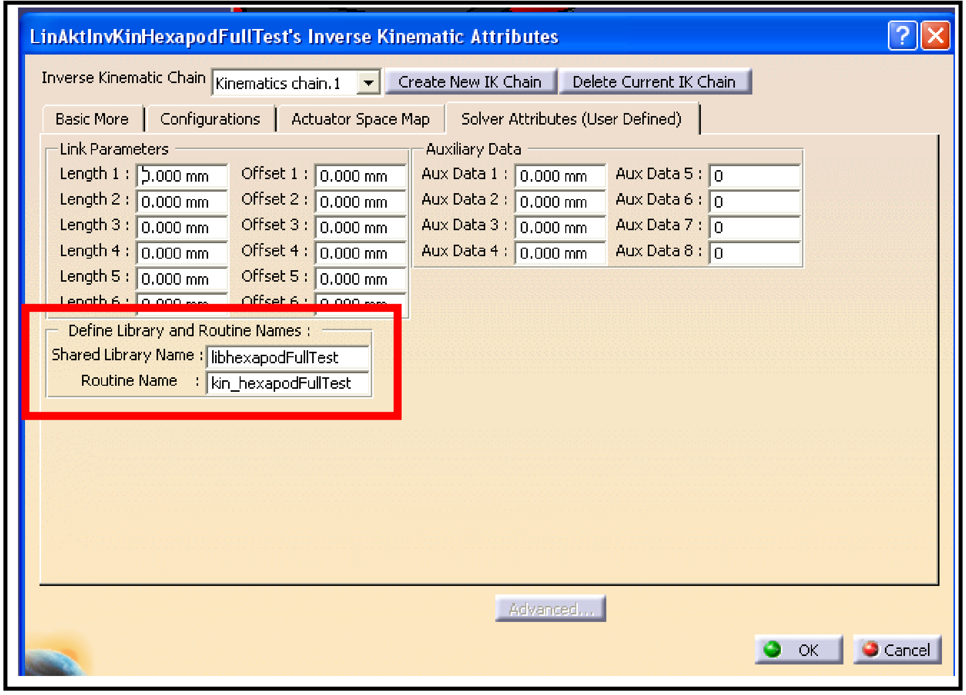


Inverse Kinematic Chain: Kinematics chain.1

	Joints Map	Joints Type	Kin Axis Type	Kin DOF	Kin Part
Command 1 :	dof(1)	Translational	Trans Z	1	LinActRod.1
Command 2 :	dof(2)	Translational	Trans Z	2	LinActRod.2
Command 3 :	dof(3)	Translational	Trans Z	3	LinActRod.3
Command 4 :	dof(4)	Translational	Trans Z	4	LinActRod.4
Command 5 :	dof(5)	Translational	Trans Z	5	LinActRod.5
Command 6 :	dof(6)	Translational	Trans Z	6	LinActRod.6

Figure C.4. Defining Actuators

- The *Solver Attributes* tab has no offset or any auxiliary values since the C-file covers those values in the code. Thus, the tab should only have a change in the C-file and library names.



Inverse Kinematic Chain: Kinematics chain.1

Length 1 :	0.000 mm	Offset 1 :	0.000 mm	Aux Data 1 :	0.000 mm	Aux Data 5 :	0
Length 2 :	0.000 mm	Offset 2 :	0.000 mm	Aux Data 2 :	0.000 mm	Aux Data 6 :	0
Length 3 :	0.000 mm	Offset 3 :	0.000 mm	Aux Data 3 :	0.000 mm	Aux Data 7 :	0
Length 4 :	0.000 mm	Offset 4 :	0.000 mm	Aux Data 4 :	0.000 mm	Aux Data 8 :	0
Length 5 :	0.000 mm	Offset 5 :	0.000 mm				
Length 6 :	0.000 mm	Offset 6 :	0.000 mm				

Shared Library Name : libhexapodFullTest

Routine Name : kin_hexapodFullTest

Figure C.5. Defining C-file and library names

THE C-FILE FOR HEXAPOD

```
/*****************************************************************************
***

**
**                      USER KINEMATICS EXAMPLE
**
**  Copyright (c) 1990 Delmia Corporation, All rights reserved.
**
**  This file contains an example of a kinematics routine for the
**  shared library.  This example will work for 4 DOF 2 Config (left and
right
**  elbow) scara robots such as the ASEA/IRB300.  By default,
**  kin_usr1 is mapped to this routine.
**
**  For a description of kinematics solutions refer to:
**
**      Paul, Richard P., "Robot Manipulators: Mathematics, Programming
**      and Control", The MIT Press, Cambridge, Massachusetts, 1981.
**
**  DESCRIPTION OF ARGUMENTS
**
**  double T6[4][4]        4x4 position matrix of center of wrist. This is
**                         the goal point MINUS the tool frame and mounting
**                         plate offsets. This is the easiest point to start
**                         the inverse kinematic solution from, and is the
**                         traditional approach.
**
**                         NOTE: T6 matrix may be transposed from your usual
**                         notation.
**
**                              | nx ny nz 0 | \\
**                         T6 = | ox oy oz 0 |  > direction cosines (9)
**                              | ax ay az 0 | /
**                              | px py pz 1 | -> position terms (3)
**
**                         px = T6[3][0];
**
**  double link_lengths[]  Distance between joint axis along link length
**
**  double link_offsets[]  Offset between joint axis along joint axis
**
**                         These two arrays can be considered the Denevitt-
**                         Hartenburg variables described in Paul's book, or
**                         any convenient scheme the user desires.
**
**  double solutions[][]   A two dimensional array contains all possible
**                         solutions for robot arm. It is up to user to
**                         decide how many solutions are possible, and to
**                         provide all solutions when routine is called:
**                         elbow up, elbow down, etc.  The CONFIGS
**                         Button in IGRIP allows user to view all possible
**                         solutions and may provide insight into importance
**                         of this array.
**
```

```
**  int warnings[]         Array providing warning states for each solution
**                         such as unreachable, singular, etc. Possible
warning
**                         states are defined in include file shlibdefs.h
**                         and are:
**
**                         WARN_GOOD_SOLUTION
**                         WARN_JOINT_LIMIT_EXCEEDED
**                         WARN_UNREACHABLE
**                         WARN_SINGULAR_SOLUTION
**
**       NOTE: shlibdefs.h is automatically included by the IGRIP Shared
**             Library Make system.  For further details regarding the
building
**             of the shared library, refer to the IGRIP Motion Pipeline
**             Reference Guide
**
**
**  Words of encouragement
**
**      Writing inverse kinematics routines is a challenge. Invariably
**      you will make mistakes which later seem trivial.  Even experts on
**      the subject loathe writing a new routine.  The usual problems
**      are matching the routines view of the world with the device
**      definition.  You must check that where this routine thinks is
**      the axis origin, or the zero reference position, is the same
**      as the IGRIP device.  Also make sure that each agree upon the
positive
**      sense of direction.  These are the most common foul ups.  Next,
**      the mounting plate offset may be wrong, so when first debugging
**      your routine, set the mounting plate and tool frame offsets to
**      zero.  Next check for dropped signs in your equations.  Maybe
**      an inverse trig function is returning an angle in a different
quadrant
**      than the one you want.  Perhaps you should be using atan2 instead
**      of atan (or vice-versa).  Remember that trig and inverse trig
function
**      angles are in radians.  Also, check array indices.  Remember that
**      arrays start at zero not one, so link_4's offset is at
link_offsets[3].
**      Are you referring to T6[3][2], when you mean T6[2][3]? Remember that
**      transformation matrices may be transposed from standard text book
**      definitions.  Once you get your routine to work you will have earned
**      the title of kinematician.
**
****************************************************************************
**/

#include <shlibdefs.h>
/*
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
**
```

```
** USER SHOULD CHANGE THESE VALUES APPROPRIATELY
**                       |
**                       |
**                      \ /
**                       v                                         */

#define NUM_SOLUTIONS    1        /* Number of possible solutions  */
#define NUM_DOFS         6        /* Number of joints to be solved */

/*                       ^
**                      / \
**                       |
**                       |
** USER SHOULD CHANGE THESE VALUES APPROPRIATELY
**
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
*/
/*
 * User must supply this function
*/
DllExport int
get_kin_config( char *kin_routine, int *kin_dof, int *solution_count, int
*usrKinDataHint )
{
      if( strcmp( kin_routine, "kin_hexapodFullTest" ) == 0 )
      {
            *kin_dof = NUM_DOFS;
            *solution_count = NUM_SOLUTIONS;
                  /*
                  * this indicates kin_usr's last argument (void *pData)
                  * will be DLM_Data_KinStat
                  */
                  *usrKinDataHint = USR_KIN_DATA_KINSTAT;
                  return 0;
      }
      return 1;
}
static char JointType[2][24] = { "ROTATIONAL", "TRANSLATIONAL" };
static char KinMode[2][24] = { "Normal", "TrackTCP" };
/*
** Routine Name
*/
DllExport int
kin_hexapodFullTest(
      link_lengths,
      link_offsets,
      T6, /* See above for description of these arguments */
      solutions,
      warnings,
            pData
      )
/*
```

```
** Passed Variable Declarations
*/
double T6[4][4],
      link_lengths[],
      link_offsets[],
      solutions[][NUM_SOLUTIONS];
int warnings[];

void *pData; /* usr routine should NEVER delete pData */

{
/*
** Local Variable Declarations (add variable declarations as appropriate)
*/

long double nx, ny, nz, ox, oy, oz, ax, ay, az, px, py, pz;
long double D11, D12, D13, D21, D22, D23, D31, D32, D33, D41, D42, D43, D51,
D52, D53, D61, D62, D63;
long double L1,L2,L3,L4,L5,L6,J1,J2,J3,J4,J5,J6, Lref;

//Variables to perform matrix multiplication
int row1,row2,row3,row4,row5,row6;
int col1,col2,col3,col4,col5,col6;
int inner1,inner2,inner3,inner4,inner5,inner6;

// The upper attachmentpoints for each leg (The vector between the TCP and
each upper attachment point).
long double L1tToTCP[4][1];
long double L2tToTCP[4][1];
long double L3tToTCP[4][1];
long double L4tToTCP[4][1];
long double L5tToTCP[4][1];
long double L6tToTCP[4][1];

//The transformed T6 matrix named as TCP matrix (do not confuse with Tool
Centre Point) - see line 307.
long double TCP[4][4];

//The current position for each upper attachmentpoint (in Base coordinates).
long double L1tCur[4] = {0};
long double L2tCur[4] = {0};
long double L3tCur[4] = {0};
long double L4tCur[4] = {0};
long double L5tCur[4] = {0};
long double L6tCur[4] = {0};

//Lower attachemnt points on each leg (in Base coordinates)
long double L1b[3];
long double L2b[3];
long double L3b[3];
long double L4b[3];
long double L5b[3];
long double L6b[3];
```

```
#if 1
/*
* using pData
*/
      int i;

      DLM_Data_KinStat *pDLM_Data = (DLM_Data_KinStat *) pData;
      if( pDLM_Data )
      {
            printf( "\n\ndof_count: %d\n", pDLM_Data->dof_count );

            printf( "\njoint_types:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%s ", JointType[(pDLM_Data->joint_types)[i]] );

            printf( "\n\nkin_mode: %s\n", KinMode[pDLM_Data->kin_mode] );

            printf( "\njoint_values:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->joint_values[i] );

            printf( "\n\njnt_trvl_lmts lower:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[0][i] );

            printf( "\n\njnt_trvl_lmts upper:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[1][i] );

            printf( "\n\n" );

      }

#endif


/***--------------- Execution Begins Here ----------------------------------
***/
      /*
      ** DO NOT REMOVE THIS BLOCK OF CODE
      ** IT IS REQUIRED TO PROPERLY SET THE NUMBER OF KINEMATIC
      ** DOFS FOR THE DEVICE
      */
      if( !kin_check_definition( NUM_DOFS, NUM_SOLUTIONS ) )
      {
      /*
      ** Inconsistency between device definition and inverse
      ** kinematics routine exists. A warning message has been
      ** issued and routine aborted
      */
      return( 1 );
      }
/***---------------- User code begins here ---------------------------------
***/
```

```
//The vector between the TCP and the upper attachmentpoints for each leg
L1tToTCP[0][0] =      31; L1tToTCP[1][0] =  48.799; L1tToTCP[2][0] = -31.45;
L1tToTCP[3][0] = 1;

L2tToTCP[0][0] =     -31; L2tToTCP[1][0] =  48.799; L2tToTCP[2][0] = -31.45;
L2tToTCP[3][0] = 1;

L3tToTCP[0][0] = -57.761; L3tToTCP[1][0] =   2.447; L3tToTCP[2][0] = -31.45;
L3tToTCP[3][0] = 1;

L4tToTCP[0][0] = -26.761; L4tToTCP[1][0] = -51.246; L4tToTCP[2][0] = -31.45;
L4tToTCP[3][0] = 1;

L5tToTCP[0][0] =  26.761; L5tToTCP[1][0] = -51.246; L5tToTCP[2][0] = -31.45;
L5tToTCP[3][0] = 1;

L6tToTCP[0][0] =  57.761; L6tToTCP[1][0] =   2.447; L6tToTCP[2][0] = -31.45;
L6tToTCP[3][0] = 1;

//The lower attachmentpoints for each leg (in Base-coordinates).
L1b[0]      =        31;      L1b[1] = 118.156;       L1b[2] = 40.205;
L2b[0]      =       -31;      L2b[1] = 118.156;       L2b[2] = 40.205;
L3b[0]      =  -117.826;      L3b[1] = -32.231;       L3b[2] = 40.205;
L4b[0]      =   -86.826;      L4b[1] = -85.925;       L4b[2] = 40.205;
L5b[0]      =    86.826;      L5b[1] = -85.925;       L5b[2] = 40.205;
L6b[0]      =   117.826;      L6b[1] = -32.231;       L6b[2] = 40.205;

//Importing the current TCP values from Delmia through the T6 matrix and
putting proper context
nx = T6[0][0];
ny = T6[0][1];
nz = T6[0][2];
ox = T6[1][0];
oy = T6[1][1];
oz = T6[1][2];
ax = T6[2][0];
ay = T6[2][1];
az = T6[2][2];
px = T6[3][0];
py = T6[3][1];
pz = T6[3][2];

//Printing the current TCP values in the debug window for evaluation purposes
printf( "\nx ny nz: %12.4f ,%12.4f ,%12.4f\n", nx ,ny ,nz );
printf( "\ox oy oz: %12.4f ,%12.4f ,%12.4f\n", ox ,oy ,oz );
printf( "\ax ay az: %12.4f ,%12.4f ,%12.4f\n", ax ,ay ,az );
printf( "\px py pz: %12.4f ,%12.4f ,%12.4f\n", px ,py ,pz );

//The transforming T6 matrix from row vectors form to column vector form
TCP[0][0] =  nx; TCP[0][1] =  ox; TCP[0][2] =  ax; TCP[0][3] = px;
TCP[1][0] =  ny; TCP[1][1] =  oy; TCP[1][2] =  ay; TCP[1][3] = py;
TCP[2][0] =  nz; TCP[2][1] =  oz; TCP[2][2] =  az; TCP[2][3] = pz;
TCP[3][0] =  0;  TCP[3][1] = 0;   TCP[3][2] =  0;  TCP[3][3] = 1;
```

```
//Printing the transformed matrix TCP
printf( "\n Transformed T6 matrix - TCP matrix\n");
printf( "\nx ox ax px: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", nx ,ox ,ax, px );
printf( "\ny oy ay py: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", ny ,oy ,ay, py );
printf( "\nz oz az pz: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", nz ,oz ,az, pz );


//Calculating the current position (in x,y,z in Base coordinates) of each
upper attachment point for each leg by multiplying the transformation
// matrix TCP[4][4] with the vector between the current TCP (the T6 matrix)
and the upper attachmentpoint for each leg (LxToTCP[][])

//Calculate upper position on Leg1 (The array L1tCur)
 for (row1 = 0; row1 < 4; row1++) {
        for (col1 = 0; col1 < 1; col1++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner1 = 0; inner1 < 4; inner1++) {
                L1tCur[row1] += TCP[row1][inner1] * L1tToTCP[inner1][col1];
            }
        }
  }
 //Calculate upper position on Leg2 (The array L2tCur)
    for (row2 = 0; row2 < 4; row2++) {
        for (col2 = 0; col2 < 1; col2++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner2 = 0; inner2 < 4; inner2++) {
                L2tCur[row2] += TCP[row2][inner2] * L2tToTCP[inner2][col2];
            }
        }
  }
//Calculate upper position on Leg3 (The array L3tCur)
    for (row3 = 0; row3 < 4; row3++) {
        for (col3 = 0; col3 < 1; col3++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner3 = 0; inner3 < 4; inner3++) {
                L3tCur[row3] += TCP[row3][inner3] * L3tToTCP[inner3][col3];
            }
        }
  }
//Calculate upper position on Leg4 (The array L4tCur)
    for (row4 = 0; row4 < 4; row4++) {
        for (col4 = 0; col4 < 1; col4++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner4 = 0; inner4 < 4; inner4++) {
                L4tCur[row4] += TCP[row4][inner4] * L4tToTCP[inner4][col4];
            }
        }
  }
```

```
//Calculate upper position on Leg5 (The array L5tCur)
    for (row5 = 0; row5 < 4; row5++) {
        for (col5 = 0; col5 < 1; col5++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner5 = 0; inner5 < 4; inner5++) {
                L5tCur[row5] += TCP[row5][inner5] * L5tToTCP[inner5][col5];
            }
          //printf("%lf\t",L5tCur[row5]);
        }
  }
//Calculate upper position on Leg6 (The array L6tCur)
    for (row6 = 0; row6 < 4; row6++) {
        for (col6 = 0; col6 < 1; col6++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner6 = 0; inner6 < 4; inner6++) {
                L6tCur[row6] += TCP[row6][inner6] * L6tToTCP[inner6][col6];
            }
          //printf("%lf\t",L6tCur[row6]);
        }
  }
// Calcultates the distance between the upper and lower attachment points for
each leg.
L1 = sqrt(((pow((L1tCur[0]-L1b[0]),2)))+((pow((L1tCur[1]-
L1b[1]),2)))+((pow((L1tCur[2]-L1b[2]),2))));

L2 = sqrt(((pow((L2tCur[0]-L2b[0]),2)))+((pow((L2tCur[1]-
L2b[1]),2)))+((pow((L2tCur[2]-L2b[2]),2))));

L3 = sqrt(((pow((L3tCur[0]-L3b[0]),2)))+((pow((L3tCur[1]-
L3b[1]),2)))+((pow((L3tCur[2]-L3b[2]),2))));

L4 = sqrt(((pow((L4tCur[0]-L4b[0]),2)))+((pow((L4tCur[1]-
L4b[1]),2)))+((pow((L4tCur[2]-L4b[2]),2))));

L5 = sqrt(((pow((L5tCur[0]-L5b[0]),2)))+((pow((L5tCur[1]-
L5b[1]),2)))+((pow((L5tCur[2]-L5b[2]),2))));

L6 = sqrt(((pow((L6tCur[0]-L6b[0]),2)))+((pow((L6tCur[1]-
L6b[1]),2)))+((pow((L6tCur[2]-L6b[2]),2))));


//The distance between upper and lower leg attachmentpoint when the command
joint is zero. Used as a reference to get the current leg length
Lref = 399.413;

//Calculates the joint values by calulating the differnce in distance between
the two attachmentpoints on each leg and a reference length (Lref)
//(the lenght between attachment points when the joints are 0)
J1 =  L1 - Lref;
J2 =  L2 - Lref;
J3 =  L3 - Lref;
J4 =  L4 - Lref;
J5 =  L5 - Lref;
J6 =  L6 - Lref;
```

```
//Sending the final joint values back to the "solutions"-matrix which is the
input matrix for Delmia.
solutions[0][0] = J1;
solutions[1][0] = J2;
solutions[2][0] = J3;
solutions[3][0] = J4;
solutions[4][0] = J5;
solutions[5][0] = J6;

//Printing some of the variable values out in the debug window to ease
debugging and get an overview of what is going on
printf( "\n The leg lengths\n" );
printf( "J1 J2 J3: %12.4f ,%12.4f ,%12.4f\n", J1 ,J2 ,J3 );
printf( "J4 J5 J6: %12.4f ,%12.4f ,%12.4f\n", J4 ,J5 ,J6 );
printf( "L1 L2 L3: %12.4f ,%12.4f ,%12.4f\n", L1 ,L2 ,L3 );
printf( "L4 L5 L6: %12.4f ,%12.4f ,%12.4f\n", L4 ,L5 ,L6);

D11 = L1tCur[0]; D12 = L1tCur[1]; D13 = L1tCur[2];
D21 = L2tCur[0]; D22 = L2tCur[1]; D23 = L2tCur[2];
D31 = L3tCur[0]; D32 = L3tCur[1]; D33 = L3tCur[2];
D41 = L4tCur[0]; D42 = L4tCur[1]; D43 = L4tCur[2];
D51 = L5tCur[0]; D52 = L5tCur[1]; D53 = L5tCur[2];
D61 = L6tCur[0]; D62 = L6tCur[1]; D63 = L6tCur[2];

printf( "\n The legs' upper attachment point coordinates \n" );
printf( "\D11 D12 D13: %12.4f ,%12.4f ,%12.4f\n", D11 ,D12 ,D13 );
printf( "\D21 D22 D23: %12.4f ,%12.4f ,%12.4f\n", D21 ,D22 ,D23 );
printf( "\D31 D32 D33: %12.4f ,%12.4f ,%12.4f\n", D31 ,D32 ,D33 );
printf( "\D41 D42 D43: %12.4f ,%12.4f ,%12.4f\n", D41 ,D42 ,D43 );
printf( "\D51 D52 D53: %12.4f ,%12.4f ,%12.4f\n", D51 ,D52 ,D53 );
printf( "\D61 D62 D63: %12.4f ,%12.4f ,%12.4f\n", D61 ,D62 ,D63 );

warnings[ 0 ] = WARN_GOOD_SOLUTION;

return (0);
}
```

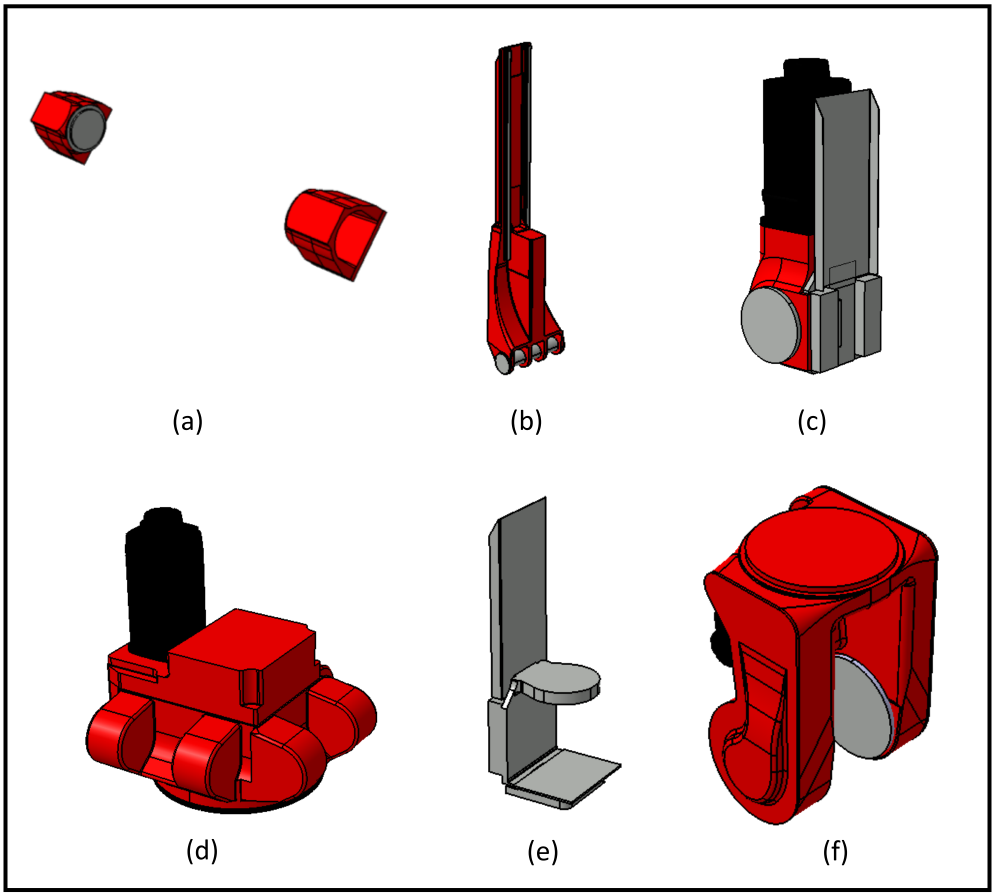
APPENDIX D: THE FORWARD AND INVERSE KINEMATICS CREATION OF EXECHON

In this section, the forward kinematics of complete structure of Exechon (including the wrist) and inverse kinematics (without the wrist attached) will be explained.

FORWARD KINEMATICS: EXECHON

The parts necessary to build an Exechon robot

a) 2 fixed base parts named *Base13* and *Base2*
b) 3 legs named *Act2* and *Act1*
c) 3 prismatic actuators *IG13* and *IG2_a*
d) 1 mobile platform named *MP*
e) 1 revolute joint component named *IG2_b* for spherical joint in leg 3
f) 1 part for the wrist's first revolute joint named *Ax4*
g) 1 part for the wrist's second revolute joint named *Ax5*
h) 1 part named *OG13* to connect 2 identical legs via revolute joints
i) 1 part named *OG2* to connect the perpendicular leg (leg 3) to base part *Base2*
j) 1 pseudo part named 0 to serve as the reference frame and base for the robot.



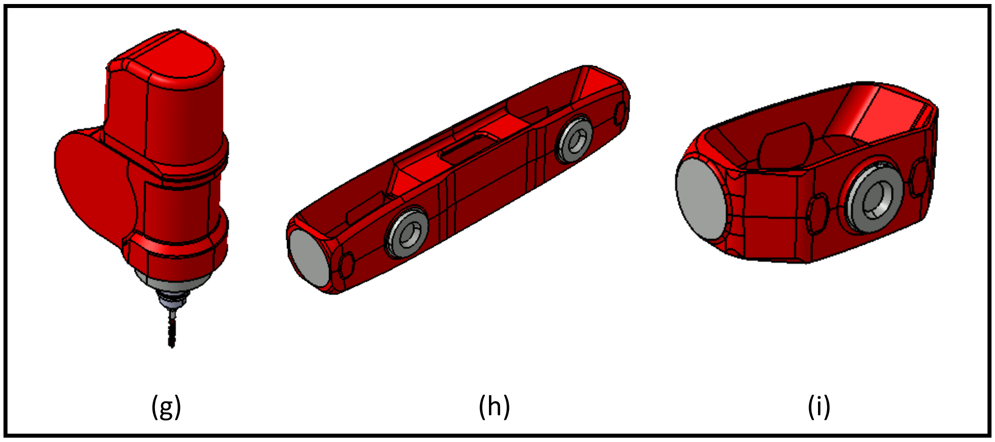
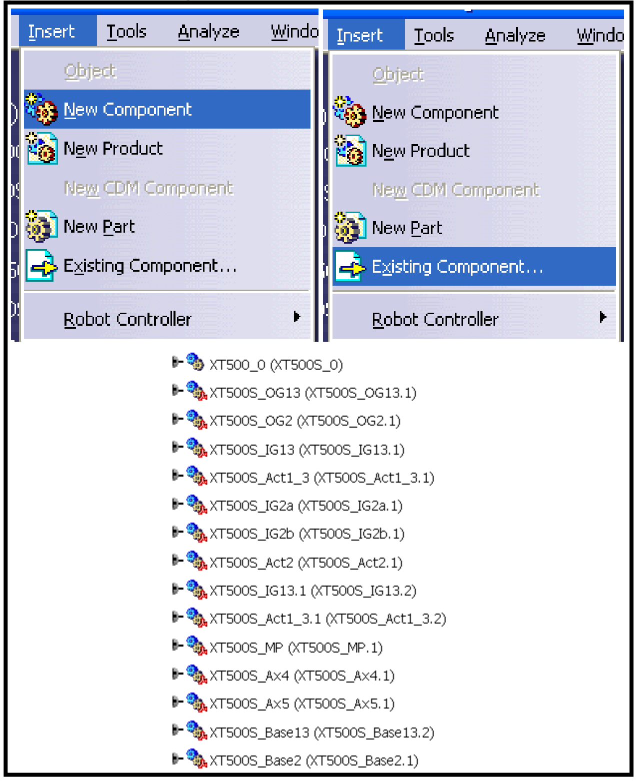


Figure D.1. Parts of Exechon

- First step in building of the mechanism is to form the node tree by first creating components using *New Component* button and inserting the necessary parts by *Existing Component* button. The steps and resulting tree should look like

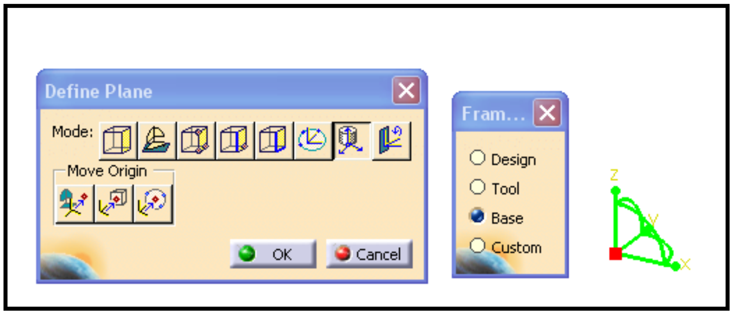


XT500_0 (XT500S_0)
XT500S_OG13 (XT500S_OG13.1)
XT500S_OG2 (XT500S_OG2.1)
XT500S_IG13 (XT500S_IG13.1)
XT500S_Act1_3 (XT500S_Act1_3.1)
XT500S_IG2a (XT500S_IG2a.1)
XT500S_IG2b (XT500S_IG2b.1)
XT500S_Act2 (XT500S_Act2.1)
XT500S_IG13.1 (XT500S_IG13.2)
XT500S_Act1_3.1 (XT500S_Act1_3.2)
XT500S_MP (XT500S_MP.1)
XT500S_Ax4 (XT500S_Ax4.1)
XT500S_Ax5 (XT500S_Ax5.1)
XT500S_Base13 (XT500S_Base13.2)
XT500S_Base2 (XT500S_Base2.1)

Figure D.2. Respective nod-tree

- The second step is to create the FOI folders as in Flexapod 6P case. Thus, click and then create the folders under each component.
- The third step is to create a new mechanism and appoint the pseudo part 0 fixed by clicking button and selecting part *0*.

- The fourth step is to place the base frame into pseudo part *0.* Click and select the FOI folder under the part *0*. For the location of FOI, select *Design* and place it in the same location at the world coordinate frame of DELMIA.



○ Design
○ Tool
● Base
○ Custom

Figure D.3. FOI for base

- The next step is to create the revolute joint between pseudo part 0 and *OG13.* First, create a FOI at the center of OG13 and another FOI at the base. The directions are then

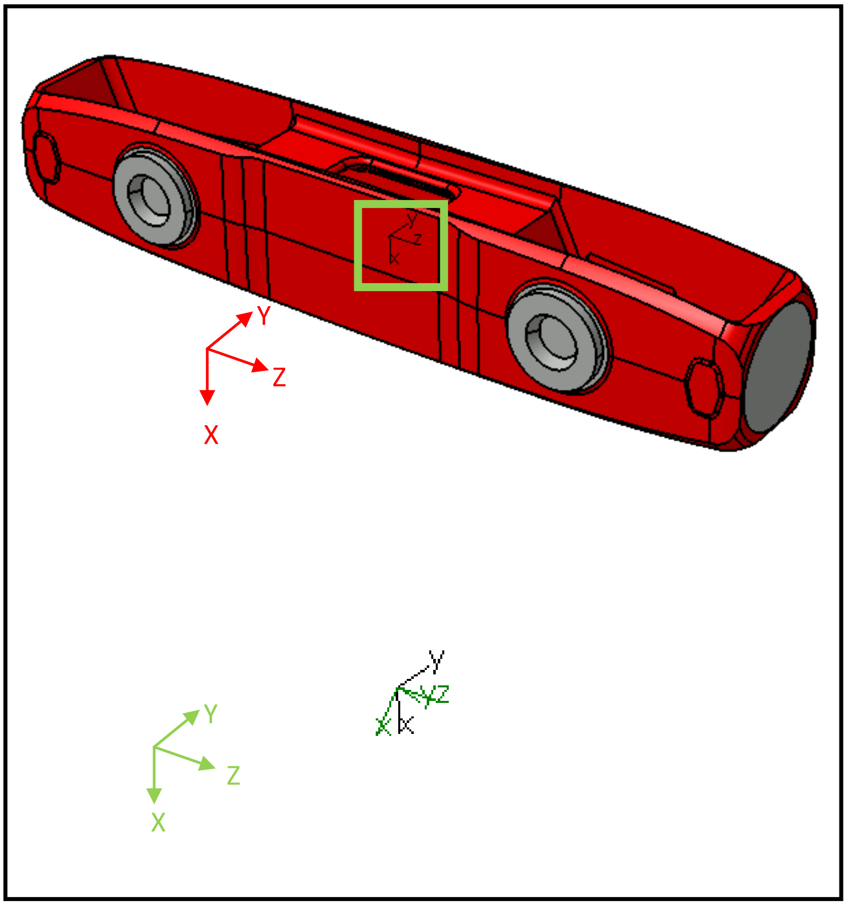
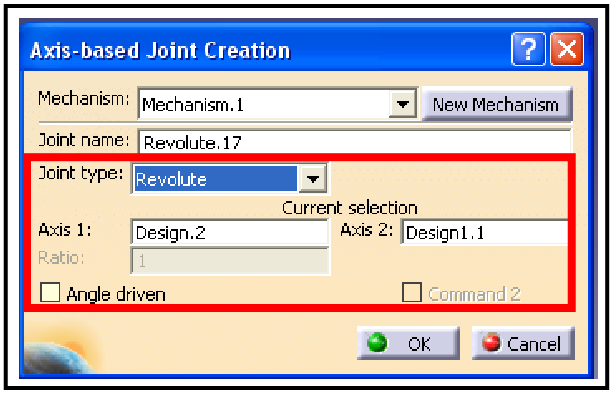


Figure D.4. FOIs for base and OG13

- Now, create a revolute joint between these FOIs by clicking button and select the revolute joint and respective FOIs. Then click *OK*.



Axis-based Joint Creation

Mechanism: Mechanism.1 New Mechanism

Joint name: Revolute.17

Joint type: Revolute

Current selection

Axis 1: Design.2 Axis 2: Design1.1

Ratio: 1

□ Angle driven □ Command 2

OK Cancel

Figure D.5. Revolute joint for base and OG13

- In this step, create a new revolute joint without actuation. First, create the respective FOIs at the center of rotation for *OG13.* Then, create the FOIs as below. The respective FOIs then

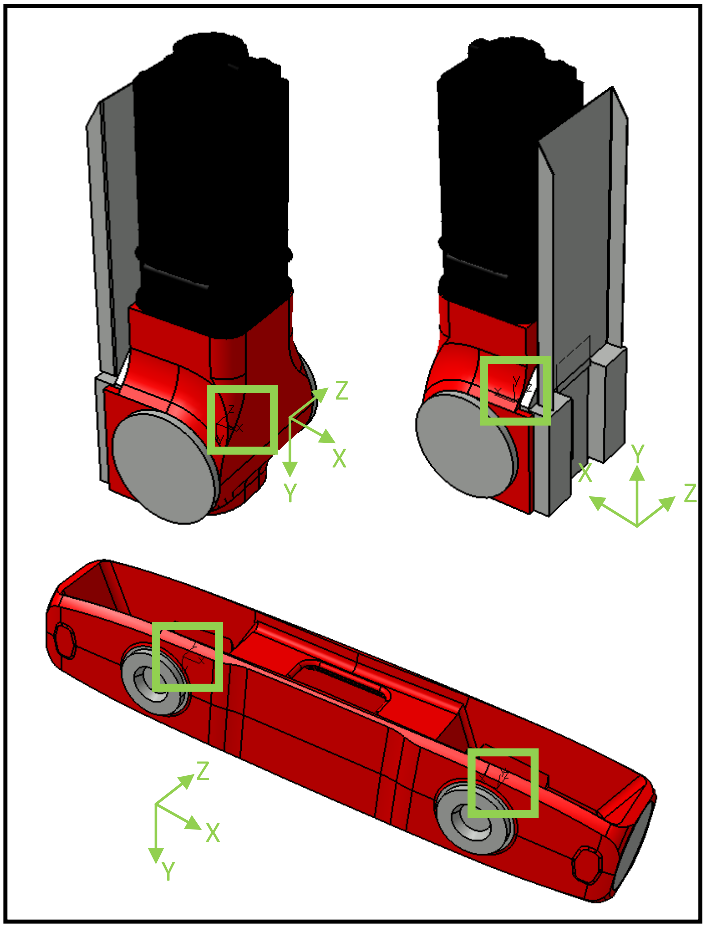


Figure D.6. FOIs for OG13 and actuators

- The result of revolute joint creation should look like

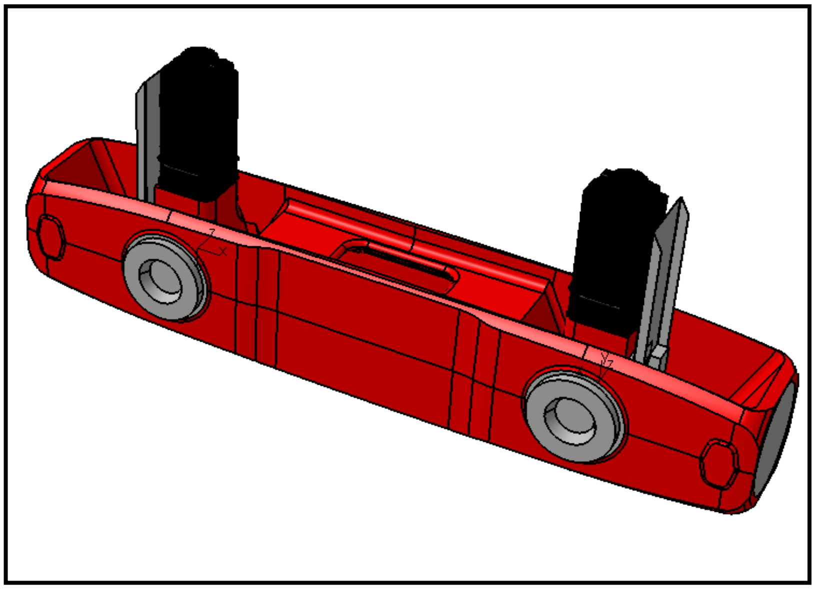


Figure D.7. Assembled actuators to OG13

- Before legs are attached, rigid joint between mobile platform and base part should be created in order to keep the prismatic joints equal to each other and compliant with standards of the robot. Thus, the required coordinates for mobile platform *MP* are [0, 50, -1250]^T.
- Along with the *MP* also bring the base part *Base12* and create another rigid joint between pseudo part *0* and *Base12.* Propagation of all the parts should be done with respect to the centers appointed while the parts are being created. The result should look like

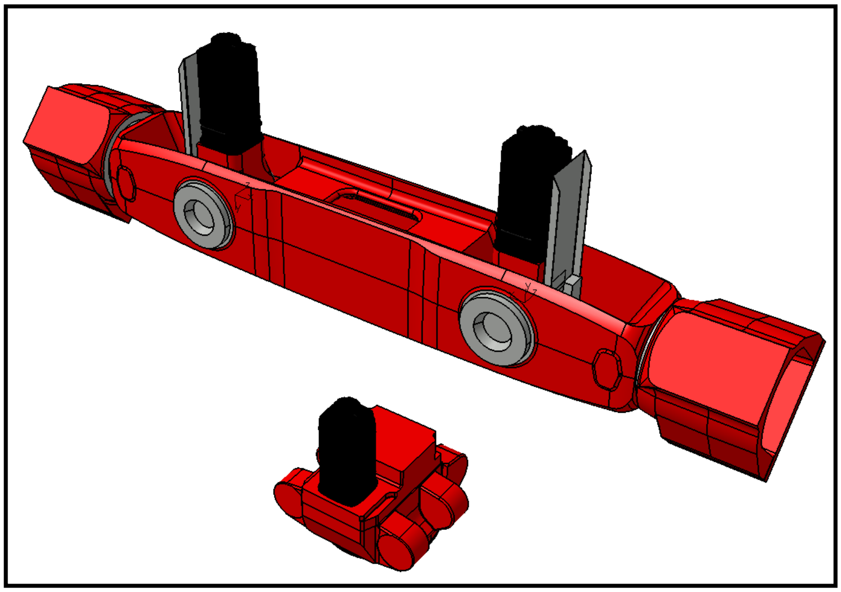


Figure D.8. Propagated mobile platform with respect to OG13

- Now, the legs 1 and 2 can be assembled via revolute joints to *MP*. First, appoint FOIs (for prismatic joints) to the centers of rotation at lower attachment points for both legs as shown in the next figure.

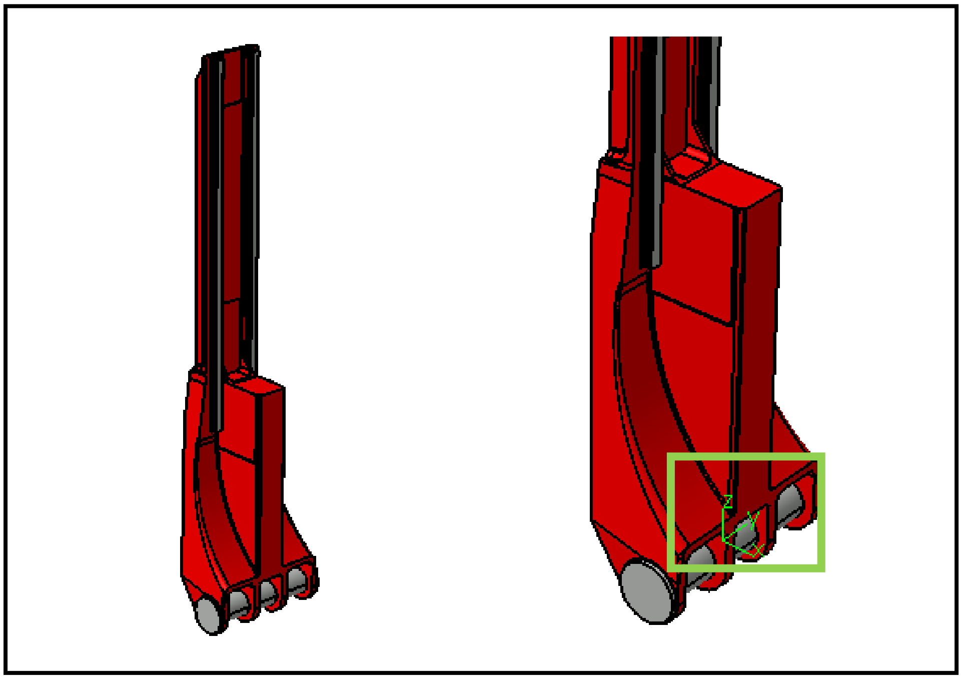


Figure D.9. FOI for prismatic joint

- Next, appoint another set of FOIs for revolute joints to both legs as

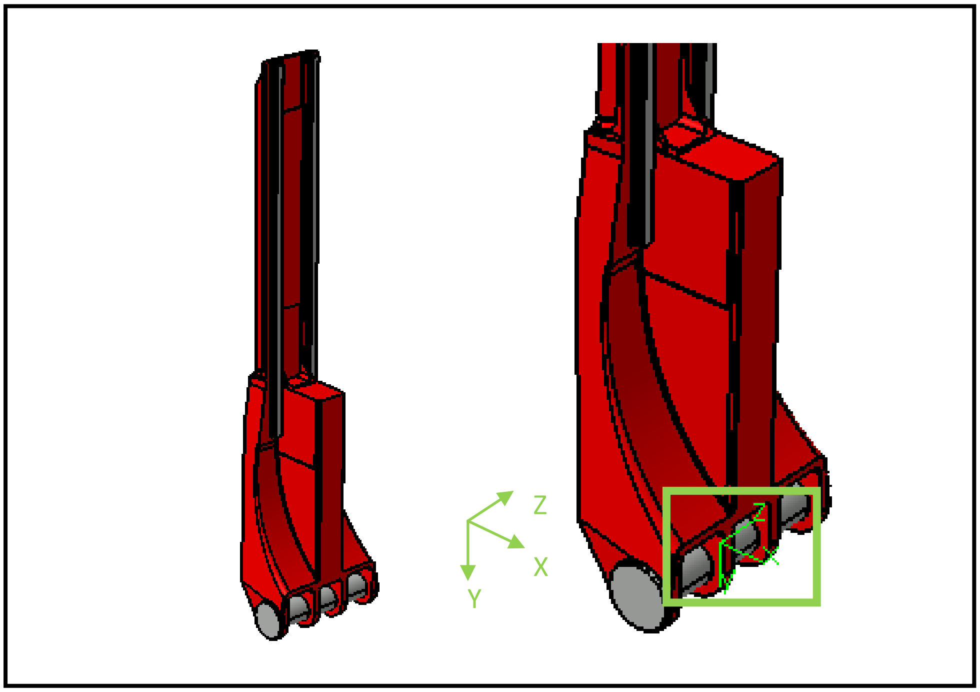


Figure D.10. FOI for revolute joint

- The second set of FOIs will be created for *MP's* revolute joints. The respective FOIs should look like

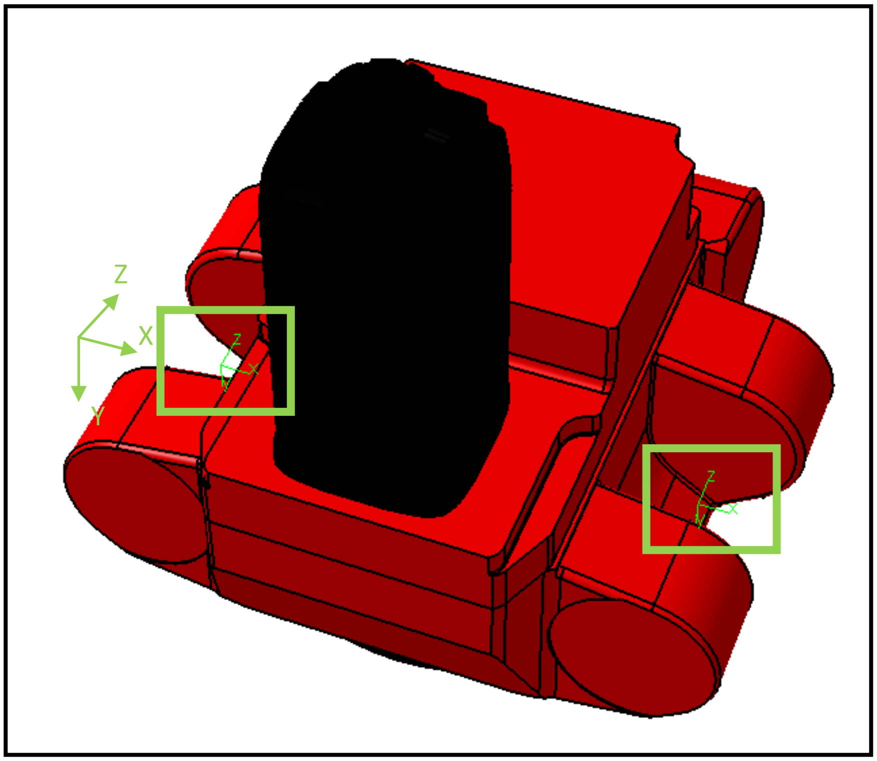


Figure D.11. FOI for mobile platform

- Now, create the revolute joints for each leg, and the result should look like

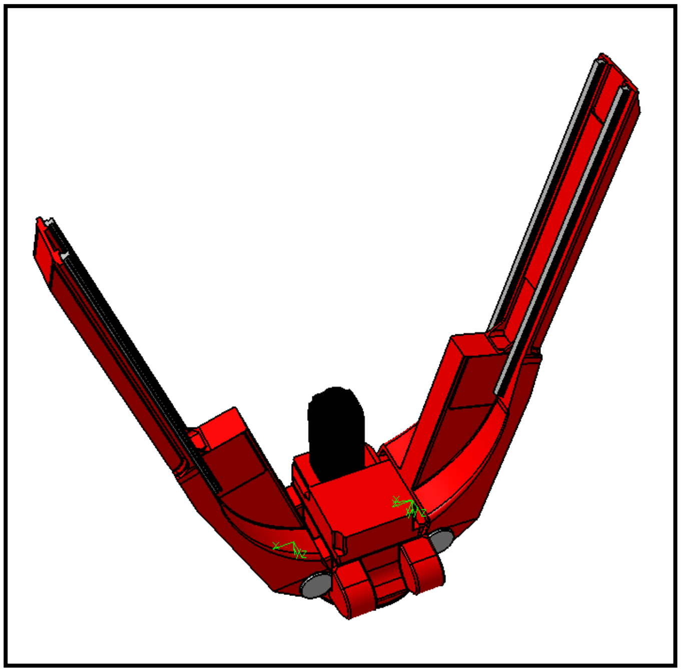


Figure D.12. Leg 1 and 2 assembled to mobile platform

- In this step, the prismatic joints will be created. First, create a set of FOIs at actuators. The FOIs must be at the center of rotation and should look like

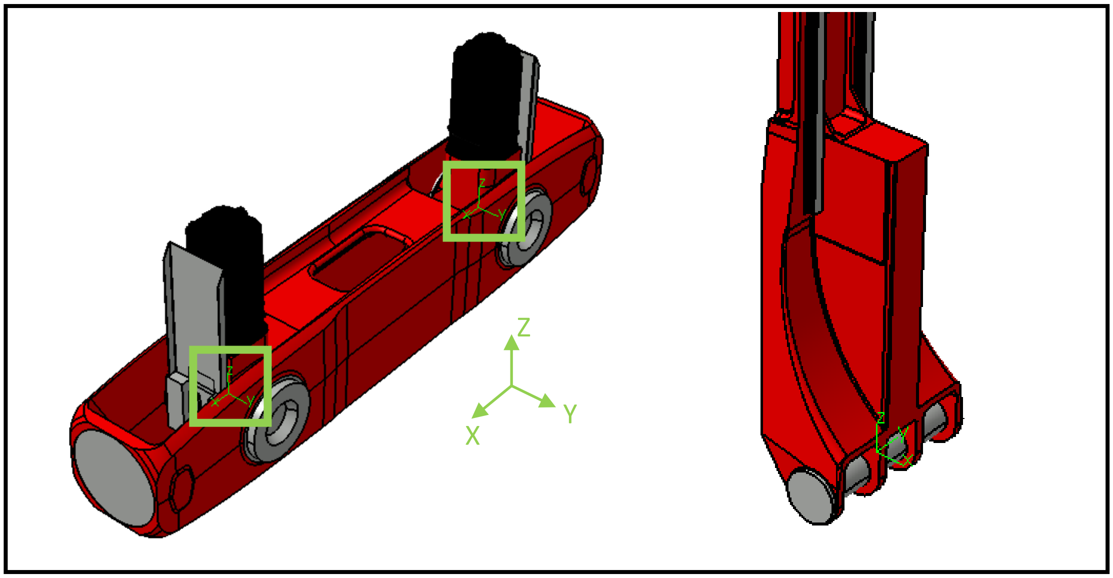


Figure D.13. FOIs in actuators for prismatic joints

- The tricky point is to directly create the prismatic joints without making any changes in the rotation of legs and the actuators. This convenient method is provided by the fact that *MP* is fixed to the base. Thus, when prismatic joints are created the result is

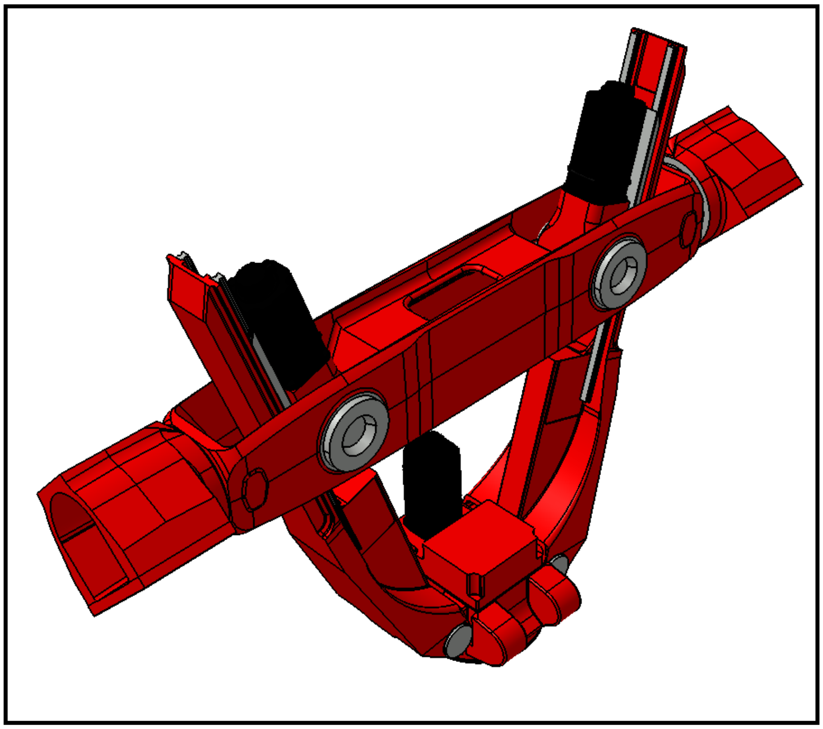


Figure D.14. Leg 1 and 2 assembled to actuators

- Now, the third leg will be connected to *MP.* The FOIs for the lower attachment points are the same as the leg 1 and 2. The FOI on *MP* will be the same as others as well. Thus, the result of FOI creation for *MP* and leg 3 should look like

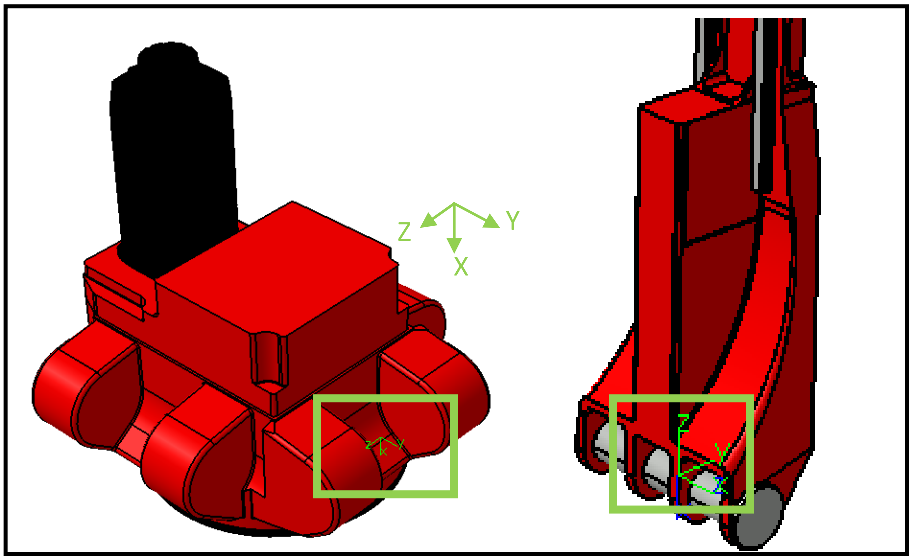


Figure D.15. FOIs for Leg 3 and mobile platform

- When the revolute joint is created, the result is then

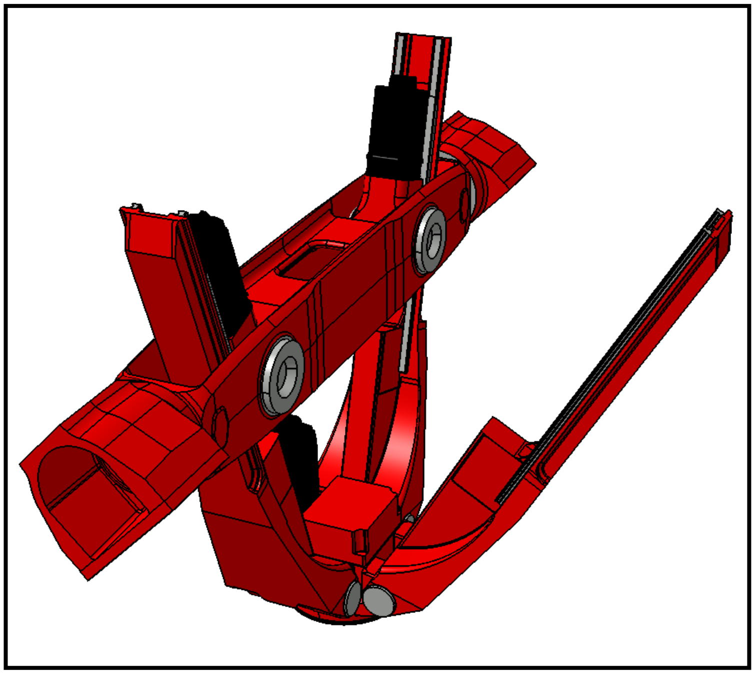


Figure D.16. Leg 3 and mobile platform assembled

- Now, the creation of the spherical joint of leg 2 will be done. It is important to start from the second base because attaching successively prismatic joints first and the spherical joint later may result in a state where standards for prismatic joint and the second base are out of limits. Thus, first create a FOI in the pseudo part 0 at the coordinates [0, 670, 0]^T. The respective FOI should look like

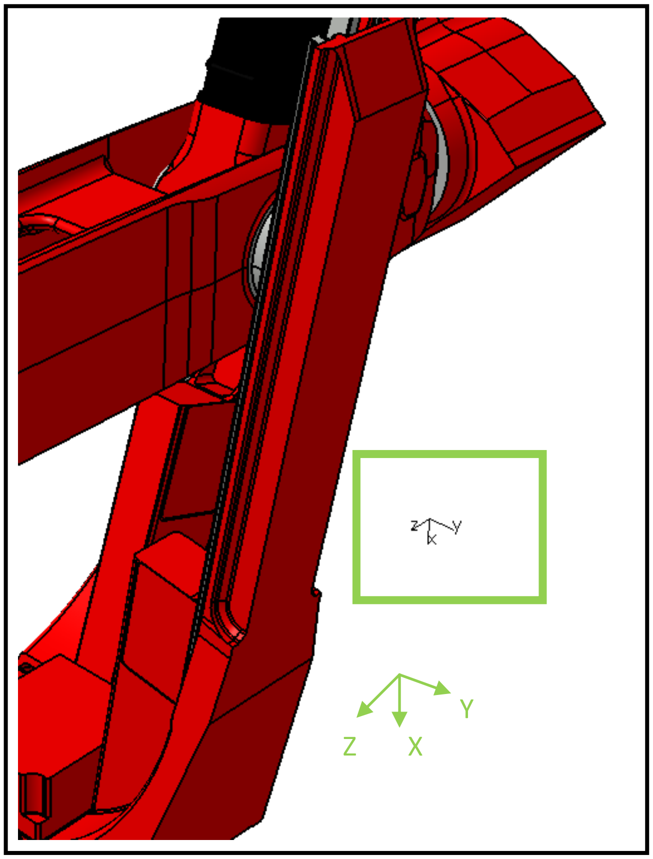


Figure D.17. FOIs for the second base

- Now, propagate *Base2* to the same location as previous FOI and create a rigid joint between pseudo part *0.* The result should look like

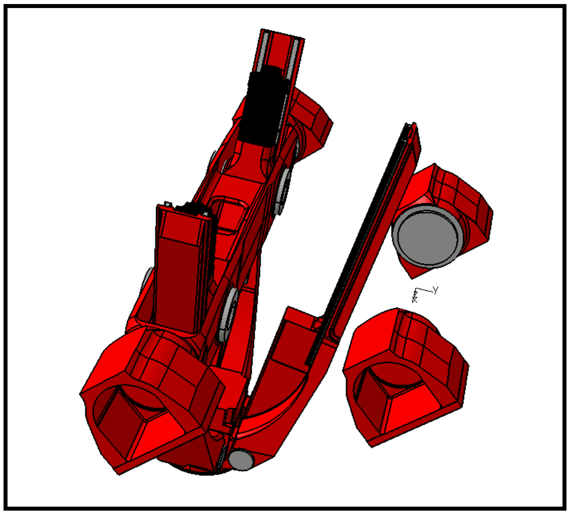


Figure D.18. Rigid joint creation for base2 with pseudo base

- In this step, the 3 successive revolute joints which correspond to a spherical joint will be built. The first revolute joint will be the rotation between *Base 2* and the *OG2.* As usual, the FOIs will be appointed first. This time, though, pseudo part *0*'s second FOI will be used since its Z-axis corresponds to the rotation of this joint. Thus, only one FOI will be created in *OG2.* The FOI and the result of joint creation should look like

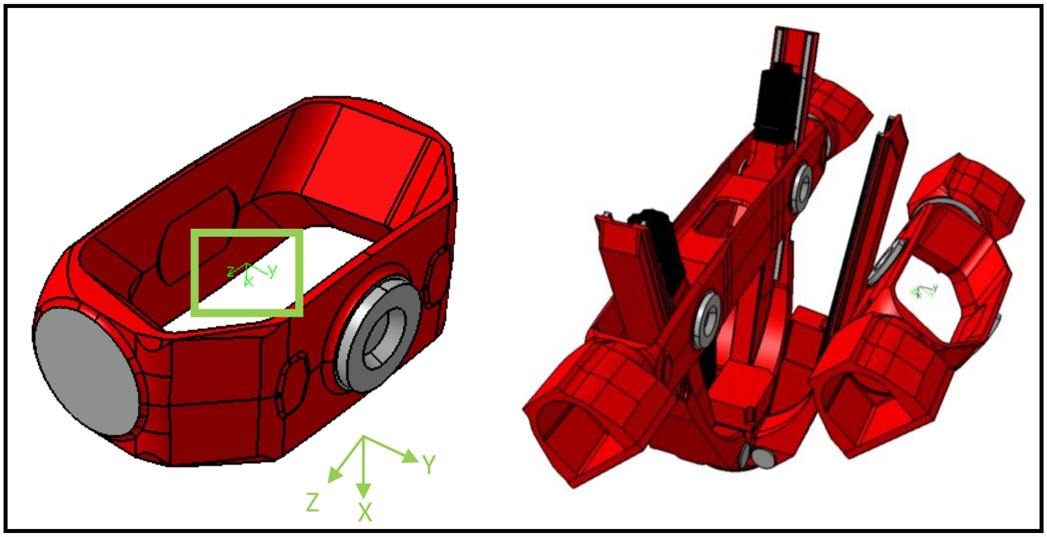


Figure D.19. FOIs for OG2 and pseudo base

- Now, the second revolute joint will be created between *OG2* and *IG2a*. This time 2 separate FOIs will be created in both parts. The FOIs should be at the center and look like

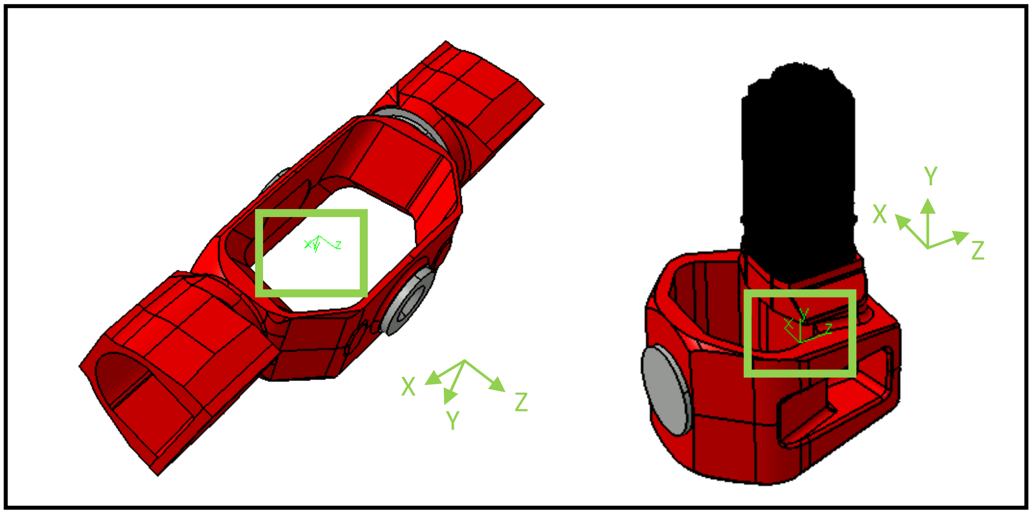


Figure D.20. FOIs for OG2 and actuator

- The result of the joint creation for the second revolute joint should look like

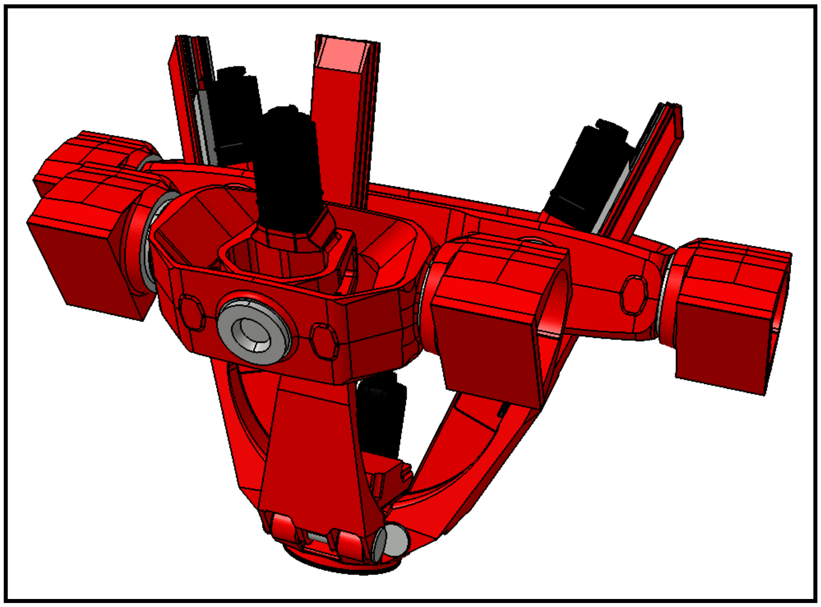


Figure D.21. Revolute joint creation between OG2 and actuator

- The last revolute joint will create the rotation between *IG2a* and *IG2b.* The respective FOIs that should be created for both parts should look like

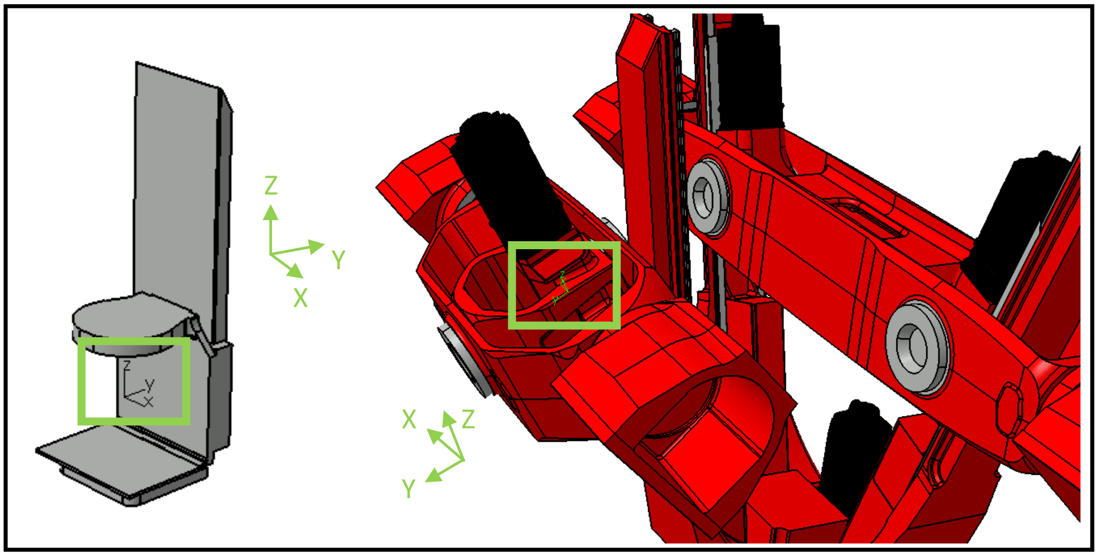


Figure D.22. FOIs for actuator components

- Then, the result of the joint creation will be

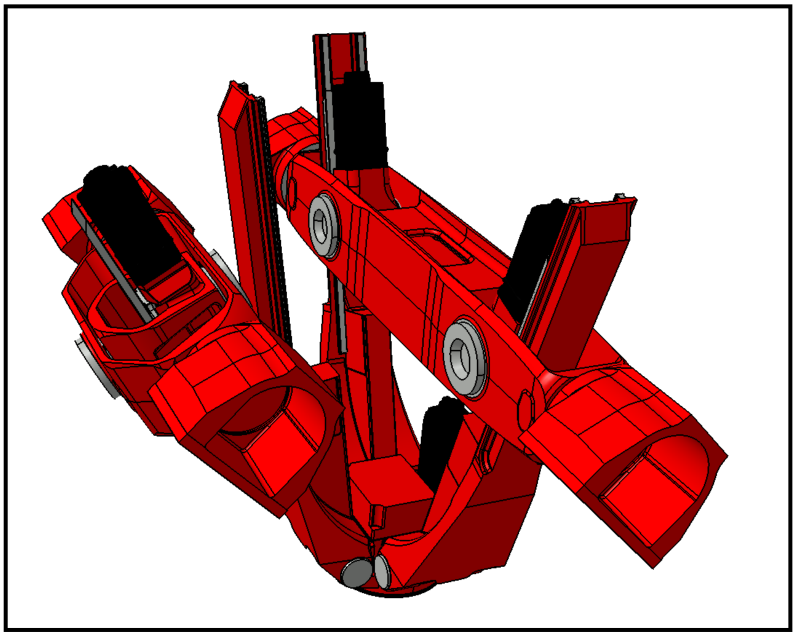
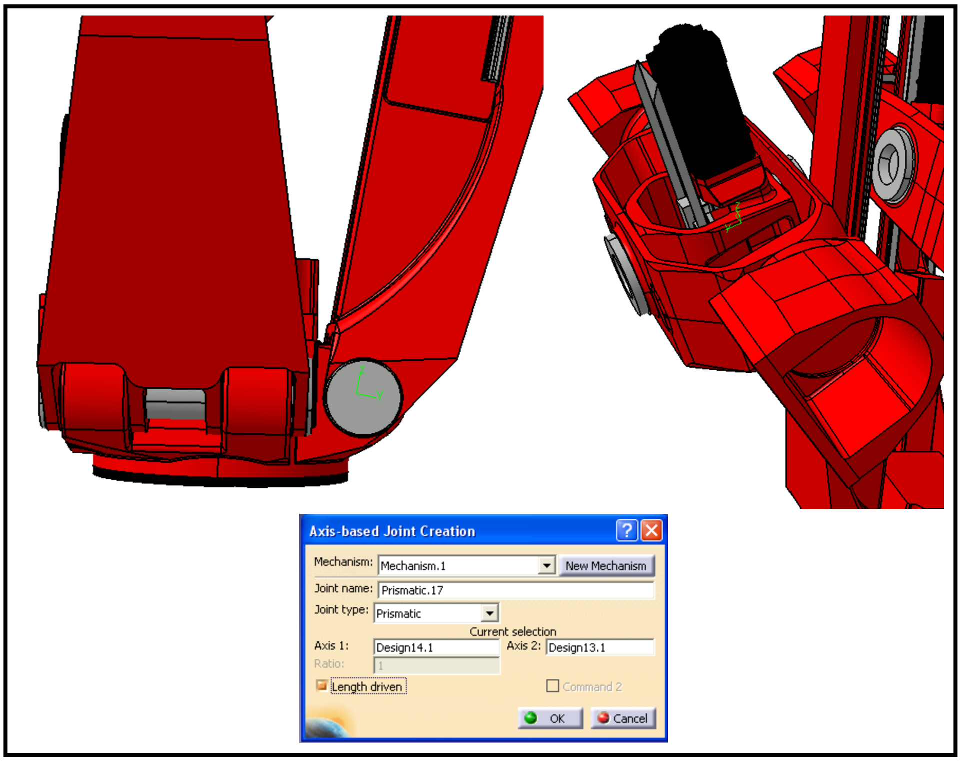


Figure D.23. Revolute joint creation for actuator components

- Now, the last joint – the prismatic – will be built. In the earlier steps, the necessary FOI for prismatic joint on the leg 3 was created. The FOI for the revolute joint used in part *IG2b* will also be used here as well. Thus, go directly the joint creation and select the respective joints. The steps should be



Axis-based Joint Creation

Mechanism: Mechanism.1

Joint name: Prismatic.17

Joint type: Prismatic

Current selection

Axis 1: Design14.1

Axis 2: Design13.1

Ratio: 1

☑ Length driven

□ Command 2

Figure D.24. Prismatic joint creation for leg and the actuator

- Hence, the result should be

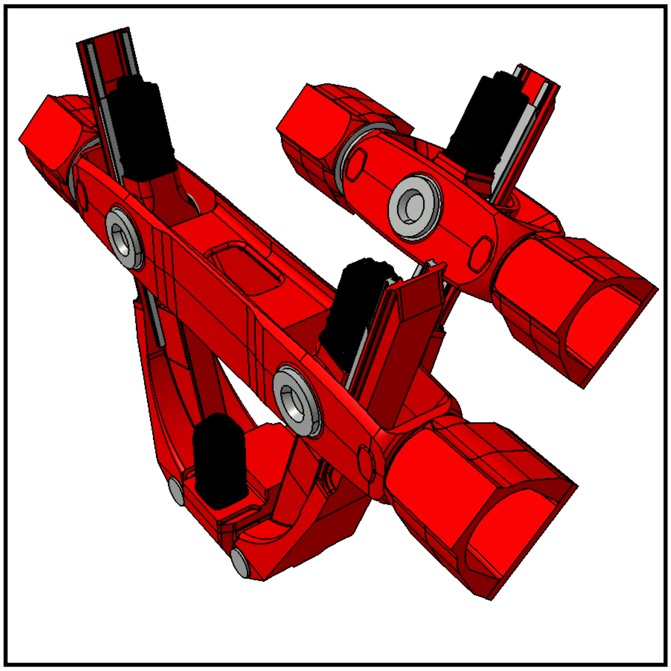


Figure D.25. Parallel structure of Exechon

- The rest of the joints will be used to build the wrist. On the other hand, the inverse kinematics will only cover for the parallel structure, which is the figure above. Thus for the joints in the wrist, a FOI in part *MP* will be created at the center of MP which also corresponds to the center of rotation for the wrist as stated in the theory section. Thus, the FOIs in *MP* and *Ax4* should look like

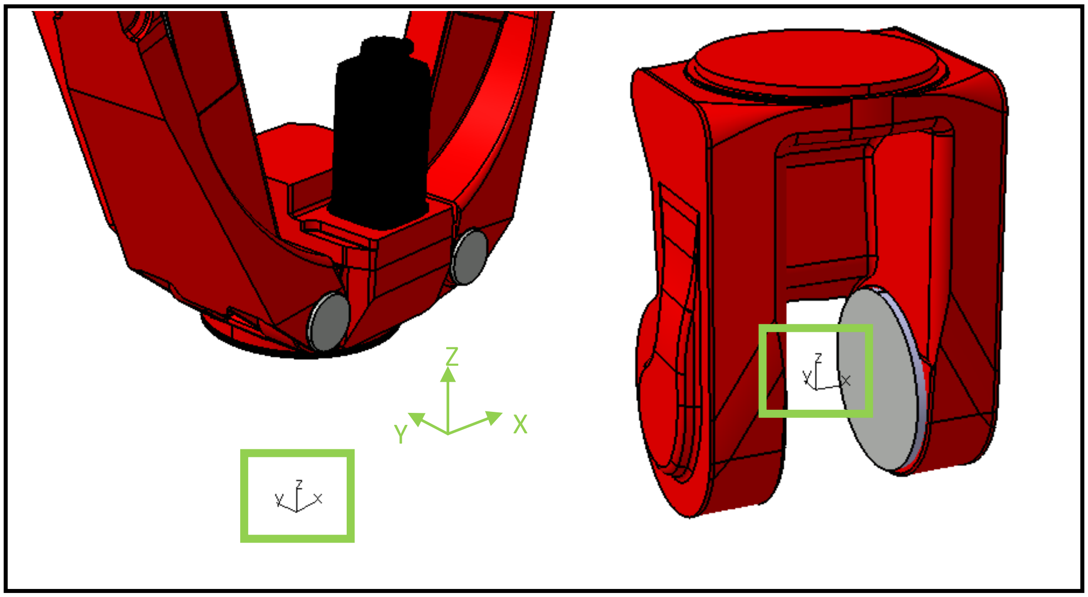


Figure D.26. FOIs for wrist and mobile platform

- The result, then, for the first revolute joint of the wrist should look like

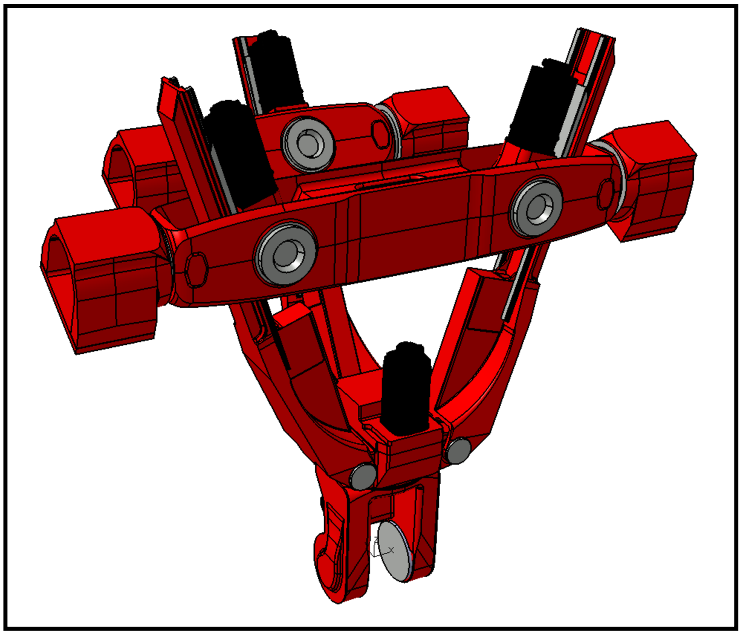


Figure D.27. Revolute joint between wrist and mobile platform

- Now the second revolute joint will be created. For this operation the respective FOIs should look like

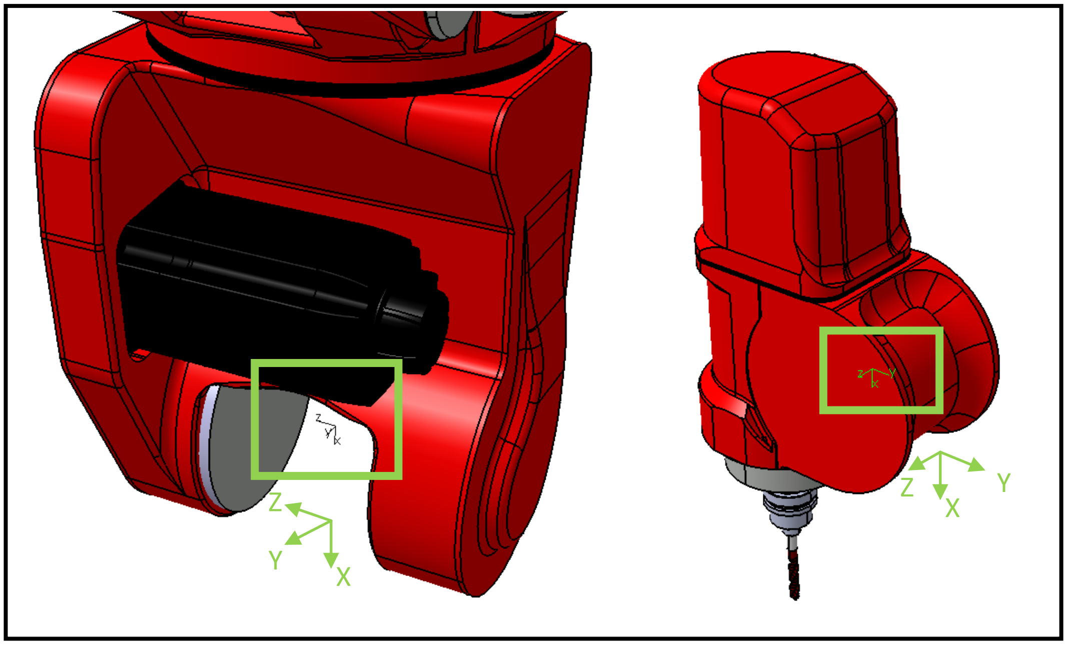


Figure D.28. FOIs for wrist and driller

- Thus, the result of the final joint creation should result as

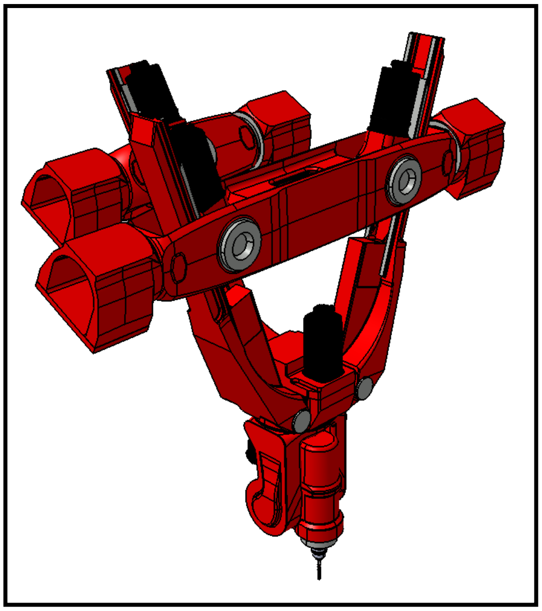


Figure D.29. Exechon’s hybrid structure

- Now, check the directions of the prismatic joints. Make sure that they all move in the same direction. This step is important when defining the joint limits. Thus, chosen directions for this thesis work are all in positive direction towards the wrist. Hence, the directions should look be as

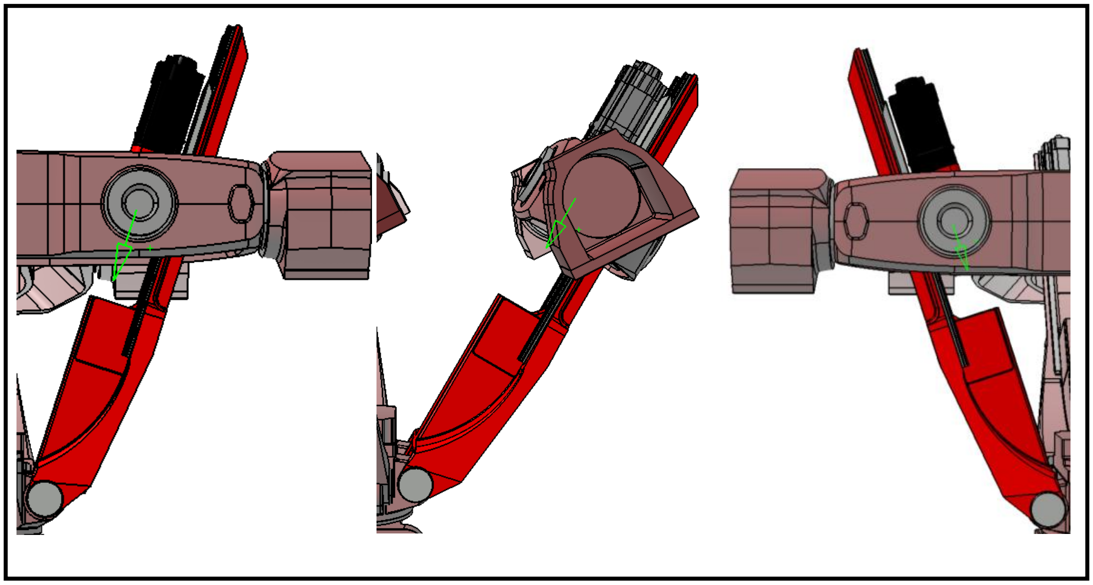
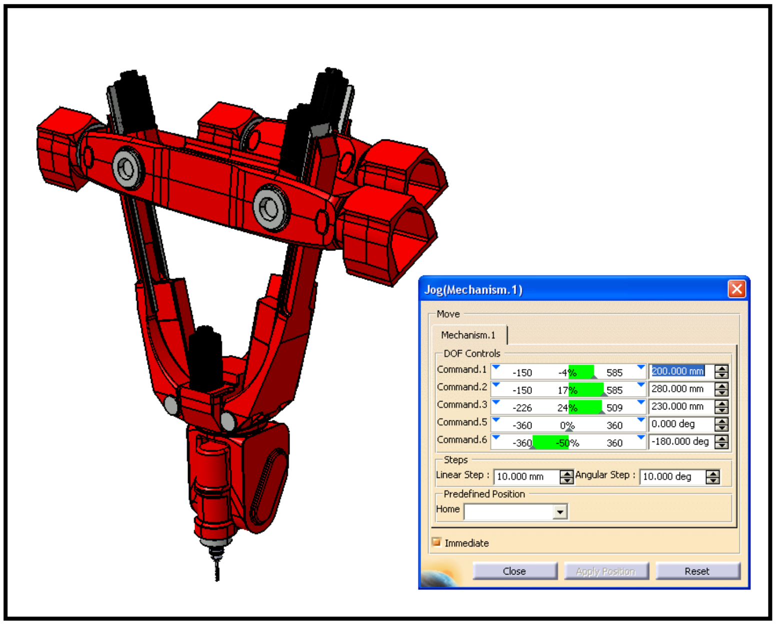


Figure D.30. Joint check

- Now, test the mechanism by using the *Jog Mechanism* property as in Flexapod 6P case. When confirmed that the system is working as intended, the measurements will be made for inverse kinematics.
- On the other hand, at some steps, DELMIA V5 might exhibit unexpected behaviors at the joints where the fixed part acts as mobile instead of the intended mobile part. In those situations, change the direction of the joint and DELMIA is most likely to respond as expected. If not, it is recommended to rebuild the system.
- When the mechanism jogged, the behavior should be as



Jog(Mechanism.1)

Move

Mechanism.1

DOF Controls

Command.1	-150	-4%	585	200.000 mm
Command.2	-150	17%	585	280.000 mm
Command.3	-226	24%	509	230.000 mm
Command.5	-360	0%	360	0.000 deg
Command.6	-360	-50%	360	-180.000 deg

Steps

Linear Step : 10.000 mm Angular Step : 10.000 deg

Predefined Position

Home

□ Immediate

Figure D.31. Motion check

- At this point, it is also important to check the posture of the joints to make sure that the complete structure is at zero position and fits the standards. To achieve this, measure the prismatic joints total lengths. Identical legs 1 and 2 should yield the same number whereas leg 3 will be different. The measurements should be done by using FOIs, and they should look like

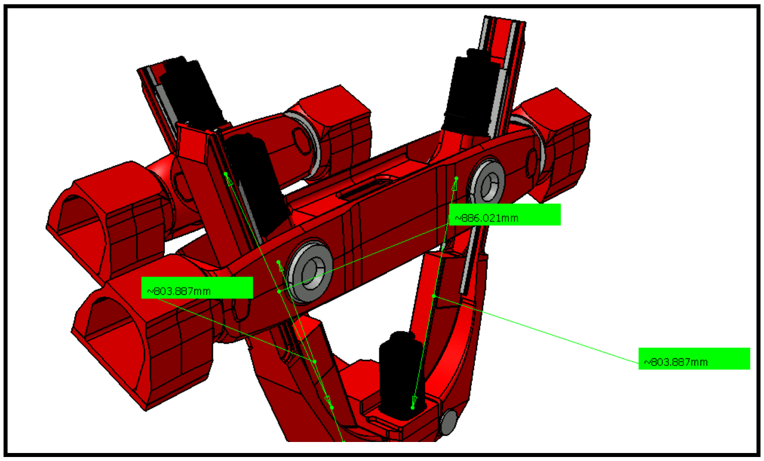


Figure D.32. Measuring travel limits

- The second set of measurements will be done to get the limits for the prismatic joints. These measurements should look like

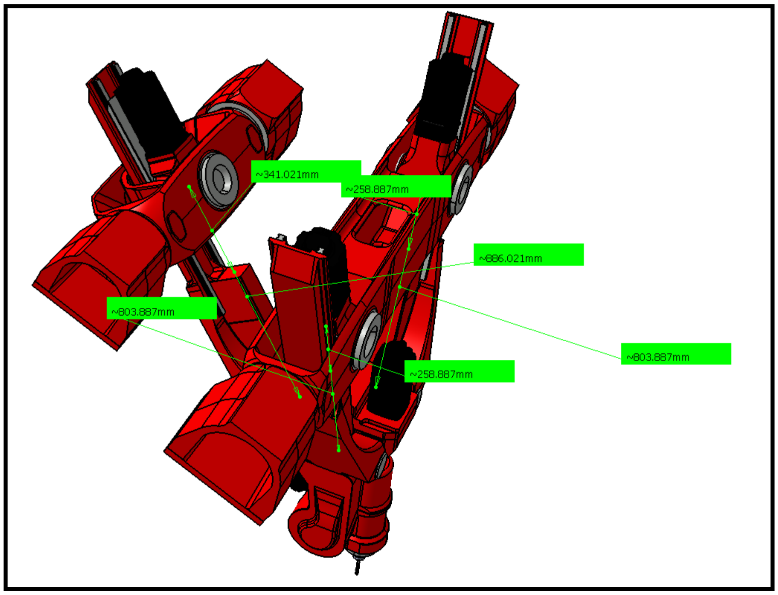
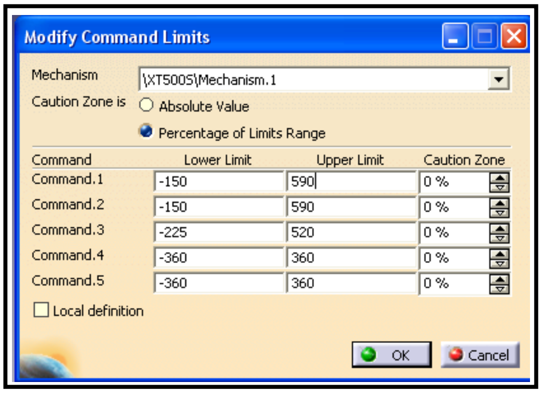


Figure D.33. Measuring other lengths to define travel limits

- Then, with the measurements the following table can be created to set the travel limits. The table then

	Prismatic Joint 1	Prismatic Joint 2	Prismatic Joint 3
Total Length	745	745	745
Lower Limit	-150	-150	-225
Upper Limit	595	595	520



Modify Command Limits

Mechanism: \XT500S\Mechanism.1

Caution Zone is ○ Absolute Value
● Percentage of Limits Range

Command	Lower Limit	Upper Limit	Caution Zone
Command.1	-150	590	0 %
Command.2	-150	590	0 %
Command.3	-225	520	0 %
Command.4	-360	360	0 %
Command.5	-360	360	0 %

□ Local definition

Figure D.34. Defining travel limits

- Since the mechanism is set and ready to use, now the inverse kinematics can be defined.

EXECHON INVERSE KINEMATICS

- As in Flexapod 6P and hexapod cases, the inverse kinematics definition will be made for DELMIA V5 environment. To start with, create a part named TCP and place it at the center of the wrist. The result should look like

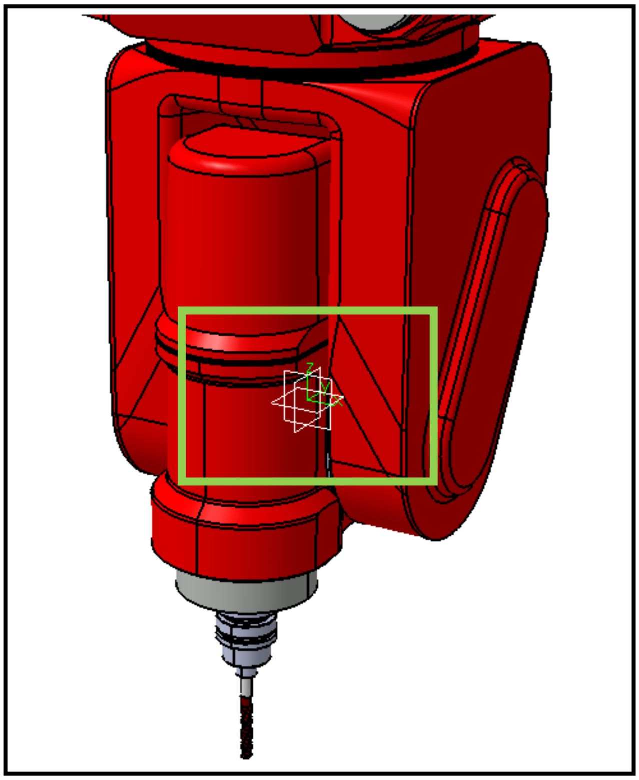
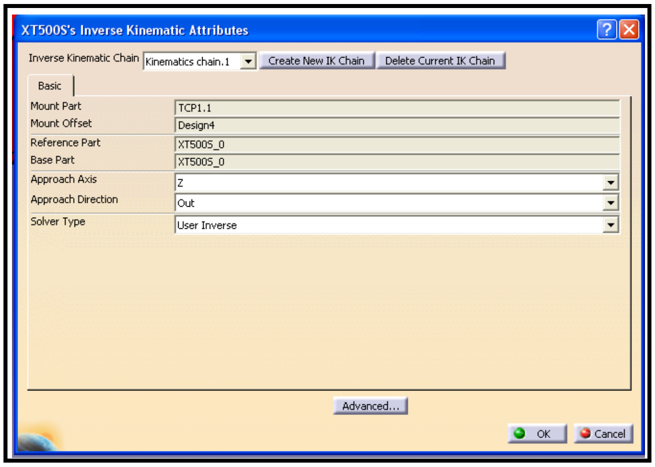


Figure D.35. Attaching new part to a TCP

- Now, click on *Inverse Kinematics* button and define the properties as following
    - **Mount Part:** TCP
    - **Mount Offset:** The FOI at the center of the wrist
    - **Base:** Pseudo part *0*
    - **Base reference:** Pseudo part *0*
    - **Approach Axis:** Z
    - **Approach Direction:** Out
    - **Solver Type:** User Inverse

- The result should look like



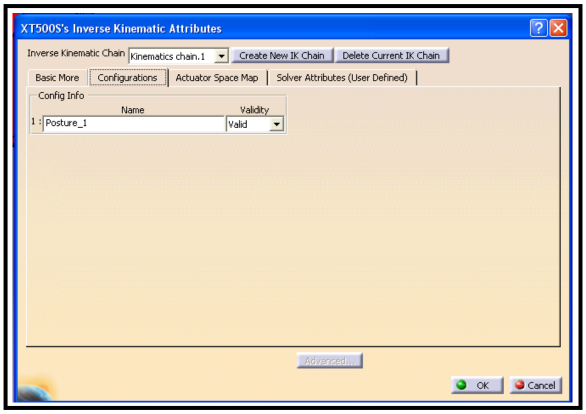
XT500S's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

Mount Part: TCP1.1
Mount Offset: Design4
Reference Part: XT500S_0
Base Part: XT500S_0
Approach Axis: Z
Approach Direction: Out
Solver Type: User Inverse

Figure D.36. Basic tab for inverse kinematics

- Now, click on *Advanced* and proceed to the *Configurations* and *Actuator Space Map* tabs. The *Configurations* tab should look like



XT500S's Inverse Kinematic Attributes

Inverse Kinematic Chain: Kinematics chain.1

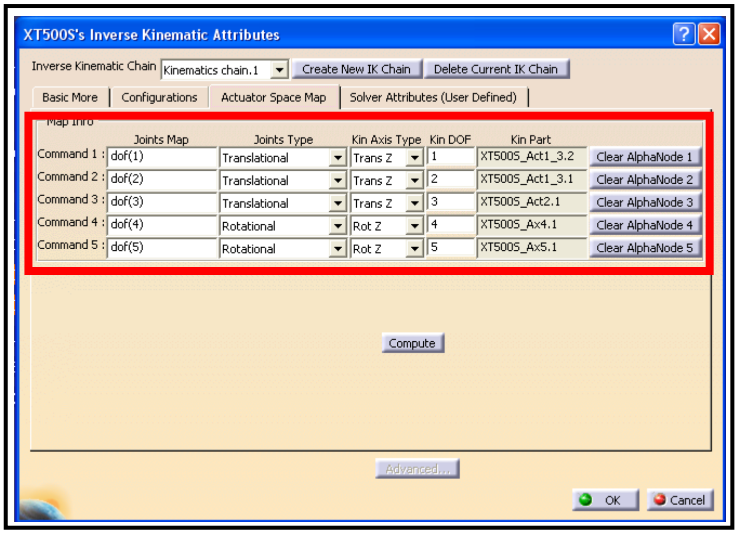
	Name	Validity
1 :	Posture_1	Valid

Figure D.37. Configurations tab for inverse kinematics

- The attributes in the *Actuator Space Map* should be as

	Joints Map	**Joints Type**	**Kin Axis Type**	**Kin DOF**	**Kin Part**
Command 1	dof(1)	Translational	Trans Z	1	Leg 1
Command 2	dof(2)	Translational	Trans Z	2	Leg 2
Command 3	dof(3)	Translational	Trans Z	3	Leg 3
Command 4	dof(4)	Rotational	Rot Z	4	Ax4
Command 5	dof(5)	Rotational	Rot Z	5	Ax5

- Then, the tab should look like



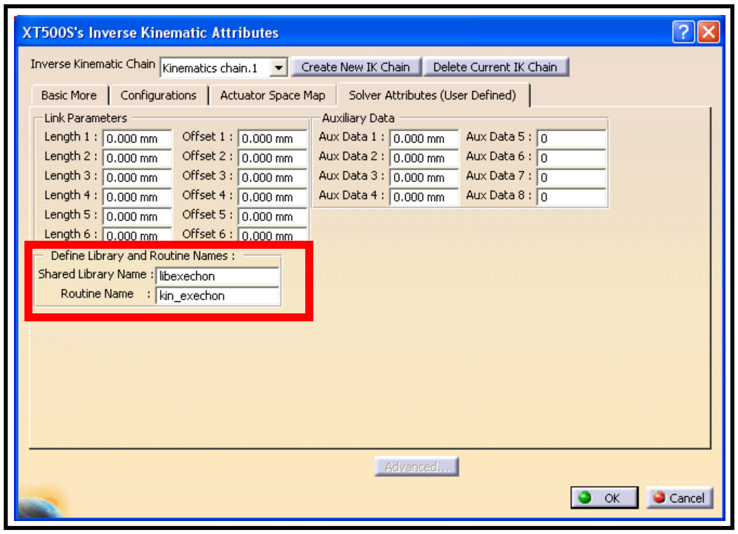
Inverse Kinematic Chain: Kinematics chain.1

Map Info

	Joints Map	Joints Type	Kin Axis Type	Kin DOF	Kin Part	
Command 1 :	dof(1)	Translational	Trans Z	1	XT500S_Act1_3.2	Clear AlphaNode 1
Command 2 :	dof(2)	Translational	Trans Z	2	XT500S_Act1_3.1	Clear AlphaNode 2
Command 3 :	dof(3)	Translational	Trans Z	3	XT500S_Act2.1	Clear AlphaNode 3
Command 4 :	dof(4)	Rotational	Rot Z	4	XT500S_Ax4.1	Clear AlphaNode 4
Command 5 :	dof(5)	Rotational	Rot Z	5	XT500S_Ax5.1	Clear AlphaNode 5

Figure D.38. Actuator parameters for inverse kinematics

- The final tab *Solver Attributes* should only contain C-file and library names



Inverse Kinematic Chain: Kinematics chain.1

Link Parameters

Length 1 :	0.000 mm	Offset 1 :	0.000 mm
Length 2 :	0.000 mm	Offset 2 :	0.000 mm
Length 3 :	0.000 mm	Offset 3 :	0.000 mm
Length 4 :	0.000 mm	Offset 4 :	0.000 mm
Length 5 :	0.000 mm	Offset 5 :	0.000 mm
Length 6 :	0.000 mm	Offset 6 :	0.000 mm

Auxiliary Data

Aux Data 1 :	0.000 mm	Aux Data 5 :	0
Aux Data 2 :	0.000 mm	Aux Data 6 :	0
Aux Data 3 :	0.000 mm	Aux Data 7 :	0
Aux Data 4 :	0.000 mm	Aux Data 8 :	0

Define Library and Routine Names :

Shared Library Name : libexechon

Routine Name : kin_exechon

Figure D.39. Library and C-file names for solver tab

- Then click *OK*, and start jogging the mechanism. As in earlier cases, compare the joint values on *Mechanism* tab to the solutions displayed on the debugging window.

EXECHON C-FILE

The C-file that should be directly transferred can be copied and pasted as seen below.

```
/*****************************************************************************
***
**
**                      USER KINEMATICS EXAMPLE
**
**  Copyright (c) 1990 Delmia Corporation, All rights reserved.
**
**  This file contains an example of a kinematics routine for the
**  shared library.  This example will work for 4 DOF 2 Config (left and
right
**  elbow) scara robots such as the ASEA/IRB300.  By default,
**  kin_usr1 is mapped to this routine.
**
**  For a description of kinematics solutions refer to:
**
**      Paul, Richard P., "Robot Manipulators: Mathematics, Programming
**      and Control", The MIT Press, Cambridge, Massachusetts, 1981.
**
**  DESCRIPTION OF ARGUMENTS
**
**  double T6[4][4]        4x4 position matrix of center of wrist. This is
**                         the goal point MINUS the tool frame and mounting
**                         plate offsets. This is the easiest point to start
**                         the inverse kinematic solution from, and is the
**                         traditional approach.
**
**                         NOTE: T6 matrix may be transposed from your usual
**                         notation.
**
**                              | nx ny nz 0 | \\
**                         T6 = | ox oy oz 0 |  > direction cosines (9)
**                              | ax ay az 0 | /
**                              | px py pz 1 | -> position terms (3)
**
**                         px = T6[3][0];
**
**  double link_lengths[]  Distance between joint axis along link length
**
**  double link_offsets[]  Offset between joint axis along joint axis
**
**                         These two arrays can be considered the Denevitt-
**                         Hartenburg variables described in Paul's book, or
**                         any convenient scheme the user desires.
**
**  double solutions[][]   A two dimensional array contains all possible
**                         solutions for robot arm. It is up to user to
**                         decide how many solutions are possible, and to
**                         provide all solutions when routine is called:
**                         elbow up, elbow down, etc.  The CONFIGS
**                         Button in IGRIP allows user to view all possible
**                         solutions and may provide insight into importance
**                         of this array.
```

```
**
**  int warnings[]         Array providing warning states for each solution
**                         such as unreachable, singular, etc. Possible 
warning
**                         states are defined in include file shlibdefs.h
**                         and are:
**
**                         WARN_GOOD_SOLUTION
**                         WARN_JOINT_LIMIT_EXCEEDED
**                         WARN_UNREACHABLE
**                         WARN_SINGULAR_SOLUTION
**
**       NOTE: shlibdefs.h is automatically included by the IGRIP Shared
**             Library Make system.  For further details regarding the 
building
**             of the shared library, refer to the IGRIP Motion Pipeline
**             Reference Guide
**
**
**  Words of encouragement
**
**      Writing inverse kinematics routines is a challenge. Invariably
**      you will make mistakes which later seem trivial.  Even experts on
**      the subject loathe writing a new routine.  The usual problems
**      are matching the routines view of the world with the device
**      definition.  You must check that where this routine thinks is
**      the axis origin, or the zero reference position, is the same
**      as the IGRIP device.  Also make sure that each agree upon the 
positive
**      sense of direction.  These are the most common foul ups.  Next,
**      the mounting plate offset may be wrong, so when first debugging
**      your routine, set the mounting plate and tool frame offsets to
**      zero.  Next check for dropped signs in your equations.  Maybe
**      an inverse trig function is returning an angle in a different 
quadrant
**      than the one you want.  Perhaps you should be using atan2 instead
**      of atan (or vice-versa).  Remember that trig and inverse trig 
function
**      angles are in radians.  Also, check array indices.  Remember that
**      arrays start at zero not one, so link_4's offset is at 
link_offsets[3].
**      Are you referring to T6[3][2], when you mean T6[2][3]? Remember that
**      transformation matrices may be transposed from standard text book
**      definitions.  Once you get your routine to work you will have earned
**      the title of kinematician.
**
*****************************************************************************
**/

#include <shlibdefs.h>
```

```
/*
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
**
** USER SHOULD CHANGE THESE VALUES APPROPRIATELY
**                        |
**                        |
**                       \ /
**                        v                                         */

#define NUM_SOLUTIONS     1         /* Number of possible solutions  */
#define NUM_DOFS          3         /* Number of joints to be solved */

/*                        ^
**                       / \
**                        |
**                        |
** USER SHOULD CHANGE THESE VALUES APPROPRIATELY
**
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
*/


/*
 * User must supply this function
*/

DllExport int
get_kin_config( char *kin_routine, int *kin_dof, int *solution_count, int
*usrKinDataHint )
{
      if( strcmp( kin_routine, "kin_exechon" ) == 0 )
      {
            *kin_dof = NUM_DOFS;
            *solution_count = NUM_SOLUTIONS;
                  /*
                  * this indicates kin_usr's last argument (void *pData)
                  * will be DLM_Data_KinStat
                  */
                  *usrKinDataHint = USR_KIN_DATA_KINSTAT;
                  return 0;
      }
      return 1;
}
```

```
static char JointType[2][24] = { "ROTATIONAL", "TRANSLATIONAL" };
static char KinMode[2][24] = { "Normal", "TrackTCP" };

/*
** Routine Name
*/
DllExport int
kin_exechon(
      link_lengths,
      link_offsets,
      T6, /* See above for description of these arguments */
      solutions,
      warnings,
            pData
      )


/*
** Passed Variable Declarations
*/
double T6[4][4],
      link_lengths[],
      link_offsets[],
      solutions[][NUM_SOLUTIONS];
int warnings[];

void *pData; /* usr routine should NEVER delete pData */

{
/*
** Local Variable Declarations (add variable declarations as appropriate)
*/
long double nx, ny, nz, ox, oy, oz, ax, ay, az, px, py, pz;
long double D11, D12, D13, D21, D22, D23, D31, D32, D33;
long double L1,L2,L3,J1,J2,J3, Lref12,Lref3;

//Variables to perform matrix multiplication
int row1,row2,row3;
int col1,col2,col3;
int inner1,inner2,inner3;

// The upper attachmentpoints for each leg (The vector between the TCP and
each upper attachment point).
long double L1tToTCP[4][1];
long double L2tToTCP[4][1];
long double L3tToTCP[4][1];

//The transformed T6 matrix named as TCP matrix (do not confuse with Tool
Centre Point) - see line 307.
long double TCP[4][4];

//The current position for each upper attachmentpoint (in Base coordinates).
long double L1tCur[4] = {0};
long double L2tCur[4] = {0};
long double L3tCur[4] = {0};
```

```
//Lower attachemnt points on each leg (in Base coordinates)
long double L1b[3];
long double L2b[3];
long double L3b[3];

#if 1
/*
* using pData
*/
      int i;

      DLM_Data_KinStat *pDLM_Data = (DLM_Data_KinStat *) pData;
      if( pDLM_Data )
      {
            printf( "\n\ndof_count: %d\n", pDLM_Data->dof_count );

            printf( "\njoint_types:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%s ", JointType[(pDLM_Data->joint_types)[i]] );

            printf( "\n\nkin_mode: %s\n", KinMode[pDLM_Data->kin_mode] );

            printf( "\njoint_values:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->joint_values[i] );

            printf( "\n\njnt_trvl_lmts lower:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[0][i] );

            printf( "\n\njnt_trvl_lmts upper:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[1][i] );

            printf( "\n\n" );

      }
#endif

/***--------------- Execution Begins Here ----------------------------------
***/
      /*
      ** DO NOT REMOVE THIS BLOCK OF CODE
      ** IT IS REQUIRED TO PROPERLY SET THE NUMBER OF KINEMATIC
      ** DOFS FOR THE DEVICE
      */
      if( !kin_check_definition( NUM_DOFS, NUM_SOLUTIONS ) )
      {
      /*
      ** Inconsistency between device definition and inverse
      ** kinematics routine exists. A warning message has been
      ** issued and routine aborted
      */
      return( 1 );
      }
/***---------------- User code begins here ---------------------------------
***/
```

```
//Importing the current TCP values from Delmia through the T6 matrix and
putting proper context
nx = T6[0][0];
ny = T6[0][1];
nz = T6[0][2];
ox = T6[1][0];
oy = T6[1][1];
oz = T6[1][2];
ax = T6[2][0];
ay = T6[2][1];
az = T6[2][2];
px = T6[3][0];
py = T6[3][1];
pz = T6[3][2];

//Printing the current TCP values in the debug window for evaluation purposes
printf( "\nx ny nz: %12.4f ,%12.4f ,%12.4f\n", nx ,ny ,nz );
printf( "\ox oy oz: %12.4f ,%12.4f ,%12.4f\n", ox ,oy ,oz );
printf( "\ax ay az: %12.4f ,%12.4f ,%12.4f\n", ax ,ay ,az );
printf( "\px py pz: %12.4f ,%12.4f ,%12.4f\n", px ,py ,pz );

//The transforming T6 matrix from row vectors form to column vector form
TCP[0][0] =  nx; TCP[0][1] =  ox; TCP[0][2] =  ax; TCP[0][3] = px;
TCP[1][0] =  ny; TCP[1][1] =  oy; TCP[1][2] =  ay; TCP[1][3] = py;
TCP[2][0] =  nz; TCP[2][1] =  oz; TCP[2][2] =  az; TCP[2][3] = pz;
TCP[3][0] =   0; TCP[3][1] =   0; TCP[3][2] =   0; TCP[3][3] =  1;

//Printing the transformed matrix TCP
printf( "\n Transformed T6 matrix - TCP matrix\n");
printf( "\nx ox ax px: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", nx ,ox ,ax, px );
printf( "\ny oy ay py: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", ny ,oy ,ay, py );
printf( "\nz oz az pz: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", nz ,oz ,az, pz );

//The vector between the TCP and the upper attachmentpoints for each leg
L1tToTCP[0][0] =    173; L1tToTCP[1][0] =    -50; L1tToTCP[2][0] = 485;
L1tToTCP[3][0] = 1;

L2tToTCP[0][0] =   -173; L2tToTCP[1][0] =    -50; L2tToTCP[2][0] = 485;
L2tToTCP[3][0] = 1;

L3tToTCP[0][0] =      0; L3tToTCP[1][0] =    173; L3tToTCP[2][0] = 485;
L3tToTCP[3][0] = 1;


//The lower attachmentpoints for each leg (in Base-coordinates).
L1b[0]      =      420;       L1b[1] =   0;           L1b[2] = 0;
L2b[0]      =     -420;       L2b[1] =   0;           L2b[2] = 0;
L3b[0]      =        0;       L3b[1] = 670;           L3b[2] = 0;


//Calculating the current position (in x,y,z in Base coordinates) of each
upper attachment point for each leg by multiplying the transformation
// matrix TCP[4][4] with the vector between the current TCP (the T6 matrix)
and the upper attachmentpoint for each leg (LxToTCP[][])
```

```
//Calculate upper position on Leg1 (The array L1tCur)
 for (row1 = 0; row1 < 4; row1++) {
        for (col1 = 0; col1 < 1; col1++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner1 = 0; inner1 < 4; inner1++) {
                L1tCur[row1] += TCP[row1][inner1] * L1tToTCP[inner1][col1];
            }
        }
  }

 //Calculate upper position on Leg2 (The array L2tCur)
    for (row2 = 0; row2 < 4; row2++) {
        for (col2 = 0; col2 < 1; col2++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner2 = 0; inner2 < 4; inner2++) {
                L2tCur[row2] += TCP[row2][inner2] * L2tToTCP[inner2][col2];
            }
        }

  }

//Calculate upper position on Leg3 (The array L3tCur)
    for (row3 = 0; row3 < 4; row3++) {
        for (col3 = 0; col3 < 1; col3++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner3 = 0; inner3 < 4; inner3++) {
                L3tCur[row3] += TCP[row3][inner3] * L3tToTCP[inner3][col3];
            }
        }

  }

// Calcultates the distance between the upper and lower attachment points for
each leg.
L1 = sqrt(((pow((L1tCur[0]-L1b[0]),2)))+((pow((L1tCur[1]-
L1b[1]),2)))+((pow((L1tCur[2]-L1b[2]),2))));

L2 = sqrt(((pow((L2tCur[0]-L2b[0]),2)))+((pow((L2tCur[1]-
L2b[1]),2)))+((pow((L2tCur[2]-L2b[2]),2))));

L3 = sqrt(((pow((L3tCur[0]-L3b[0]),2)))+((pow((L3tCur[1]-
L3b[1]),2)))+((pow((L3tCur[2]-L3b[2]),2))));


//The distance between upper and lower leg attachmentpoint when the command
joint is zero. Used as a reference to get the current leg length
Lref12 = 803.887;
Lref3 = 886.021;
//Calculates the joint values by calulating the differnce in distance between
the two attachmentpoints on each leg and a reference length (Lref)
//(the lenght between attachment points when the joints are 0)
J1 =  L1 - Lref12;
J2 =  L2 - Lref12;
J3 =  L3 - Lref3;
```

```
//Sending the final joint values back to the "solutions"-matrix which is the
input matrix for Delmia.
solutions[0][0] = J1;
solutions[1][0] = J2;
solutions[2][0] = J3;

//Printing some of the variable values out in the debug window to ease
debugging and get an overview of what is going on
printf( "\n The leg lengths\n" );
printf( "J1 J2 J3: %12.4f ,%12.4f ,%12.4f\n", J1 ,J2 ,J3 );
printf( "L1 L2 L3: %12.4f ,%12.4f ,%12.4f\n", L1 ,L2 ,L3 );

D11 = L1tCur[0]; D12 = L1tCur[1]; D13 = L1tCur[2];
D21 = L2tCur[0]; D22 = L2tCur[1]; D23 = L2tCur[2];
D31 = L3tCur[0]; D32 = L3tCur[1]; D33 = L3tCur[2];
printf( "\n The legs' lower attachment point coordinates \n" );
printf( "\D11 D12 D13: %12.4f ,%12.4f ,%12.4f\n", D11 ,D12 ,D13 );
printf( "\D21 D22 D23: %12.4f ,%12.4f ,%12.4f\n", D21 ,D22 ,D23 );
printf( "\D31 D32 D33: %12.4f ,%12.4f ,%12.4f\n", D31 ,D32 ,D33 );

warnings[ 0 ] = WARN_GOOD_SOLUTION;

return (0);

}
```

APPENDIX E: GANTRY-TAU ROBOT

In this appendix, the forward and inverse kinematics of Gantry-Tau robot will be given. The C-file that can be directly copied to another file is also attached at the end of the inverse kinematics calculation.

FORWARD KINEMATICS: GANTRY-TAU

The parts that constitute this robot are

a) Three fixed beams for the prismatic joints named *Prismatic_fixed_1, 2* and *3*

b) Three mobile parts for the prismatic joints named *Prismatic_mobile_1, 2* and *3*

c) Six links named *arm_1, 2,…,6*

d) A mobile platform named *Mobile_platform*

e) Six pseudo parts to create successive rotations named *ins_sph_1,2,…,6*

f) One pseudo part to define the base coordinate frame named *Base_ref*

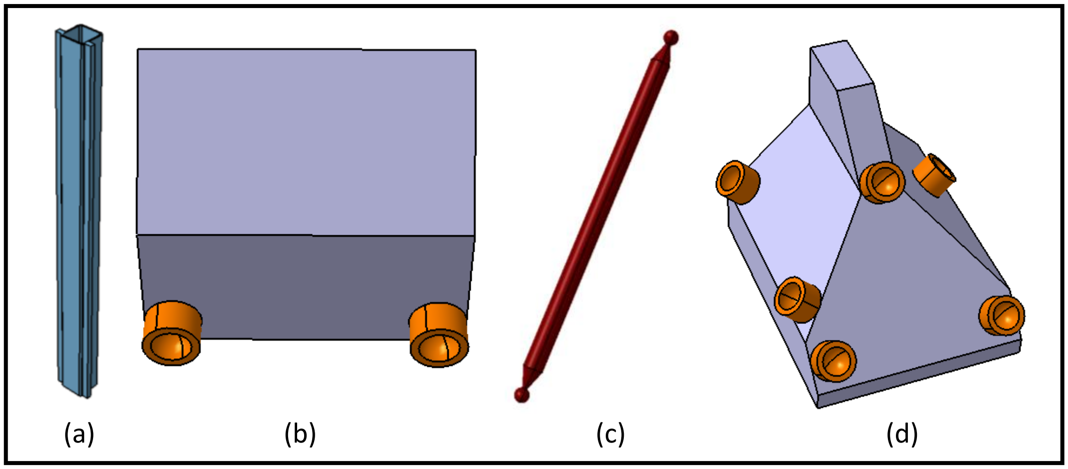
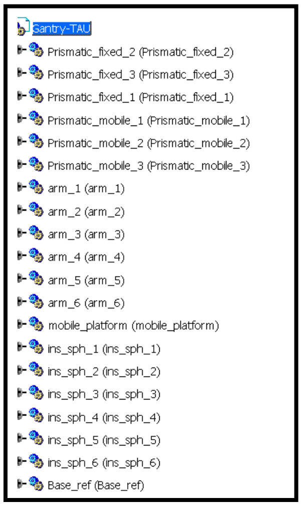


Figure E.1. Parts of Gantry-TAU

- The first step in building the mechanism is to insert the all parts as in earlier cases. The node tree then should look like



Gantry-TAU
Prismatic_fixed_2 (Prismatic_fixed_2)
Prismatic_fixed_3 (Prismatic_fixed_3)
Prismatic_fixed_1 (Prismatic_fixed_1)
Prismatic_mobile_1 (Prismatic_mobile_1)
Prismatic_mobile_2 (Prismatic_mobile_2)
Prismatic_mobile_3 (Prismatic_mobile_3)
arm_1 (arm_1)
arm_2 (arm_2)
arm_3 (arm_3)
arm_4 (arm_4)
arm_5 (arm_5)
arm_6 (arm_6)
mobile_platform (mobile_platform)
ins_sph_1 (ins_sph_1)
ins_sph_2 (ins_sph_2)
ins_sph_3 (ins_sph_3)
ins_sph_4 (ins_sph_4)
ins_sph_5 (ins_sph_5)
ins_sph_6 (ins_sph_6)
Base_ref (Base_ref)

Figure E.2. The nod-tree for Gantry-TAU

- Then, click *New Mechanism* button . Afterwards, click *Fix* button and select one of the *Prismatic_fixed* parts. From their center points, place all the beams with respect to the following coordinates:

	Prismatic_fixed_1	*Prismatic_fixed_2*	*Prismatic_fixed_3*
X	-2300	100	-1100
Y	0	0	800
Z	-3000	-3000	-3000

With the correct orientations, the result should look like

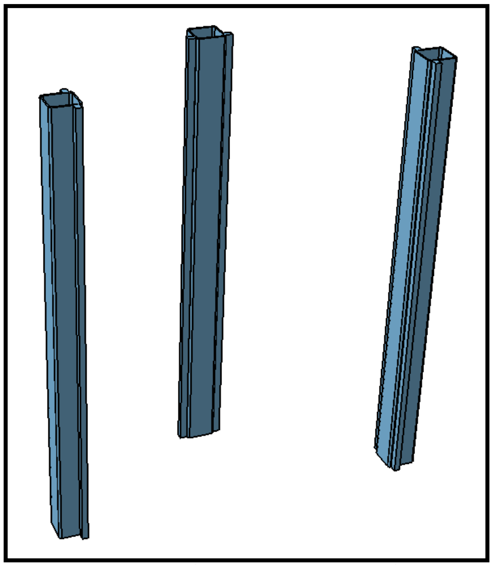


Figure E.3. The prismatic joint beams

- Then start appointing FOIs to beams for the prismatic joints with Z-directions as

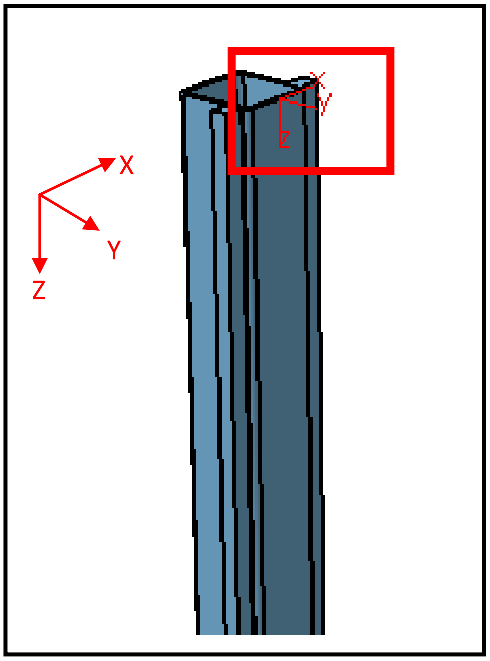


Figure E.4. FOIs for prismatic joints

- Now, appoint the corresponding FOIs to the prismatic joint's mobile parts as

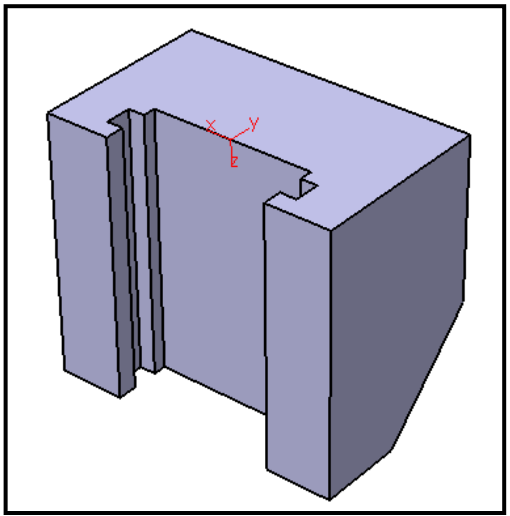


Figure E.5. FOIs for prismatic joints mobile part

- In this step, first create revolute joints between the fixed and mobile parts to make sure that the origins of each FOI are coinciding with the corresponding ones. Afterwards, delete the newly created revolute joints and directly create prismatic joints between the same FOI pairs with *Length driven* property activated. The result should look like

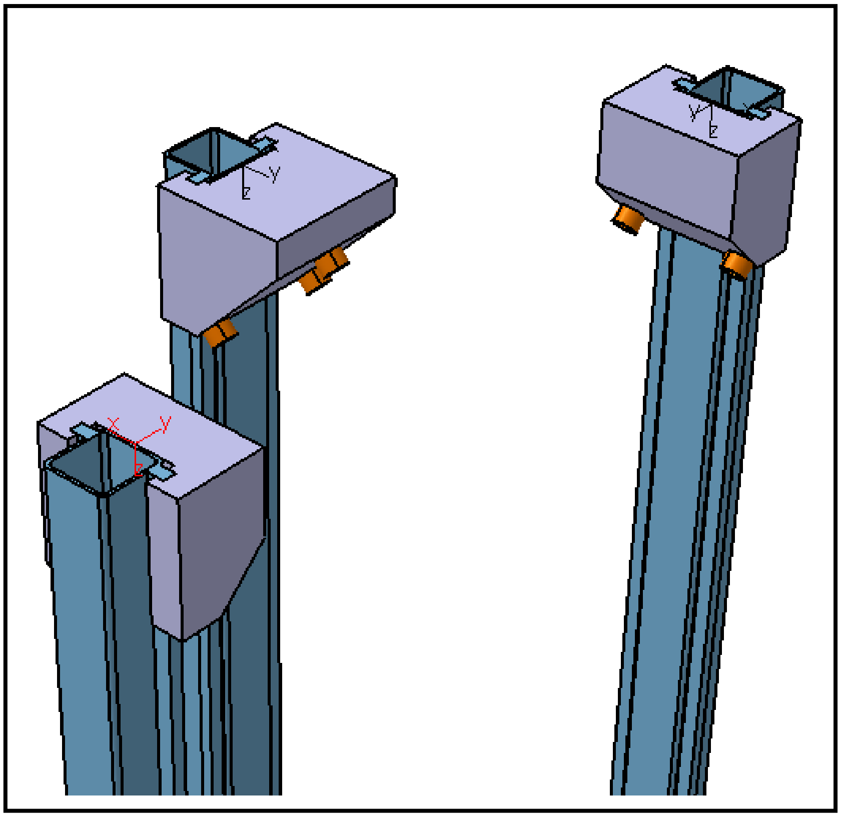


Figure E.6. Prismatic joint creation for all beams

- Now, the links will be attached to the mobile platform via spherical joints. First, create the corresponding FOIs at the center of sockets on the mobile platform. Then, create the corresponding FOIs on each link. The orientation of these set of FOIs do not matter since a spherical joint only requires a coincidence in the origin. Thus, the result should look like

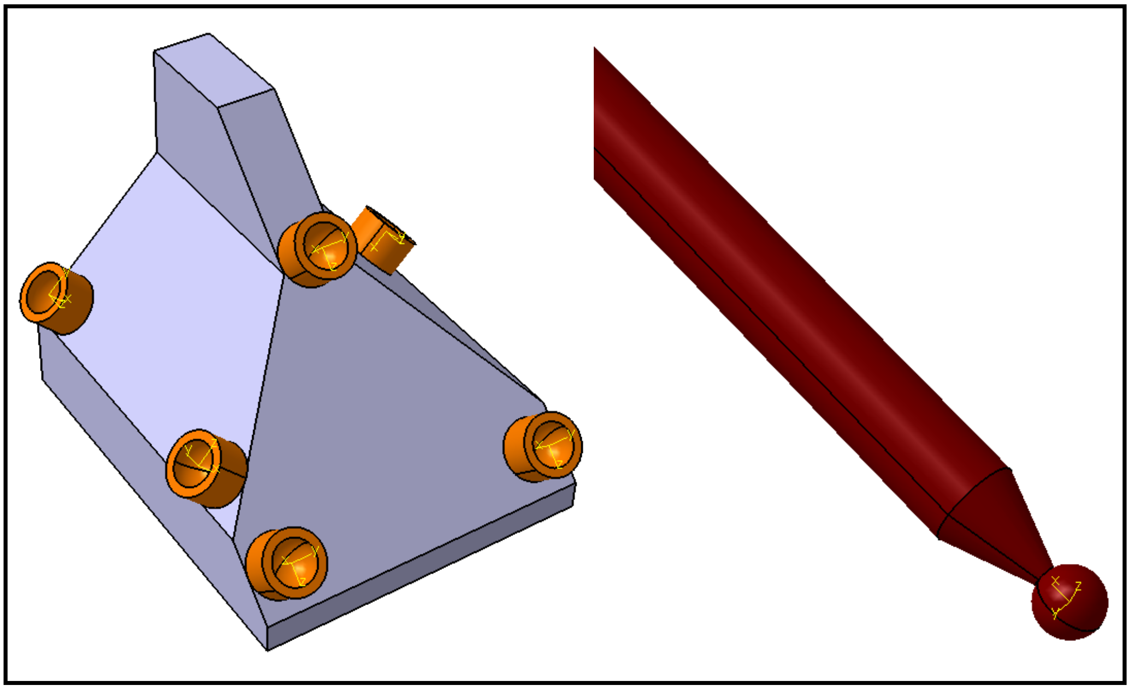
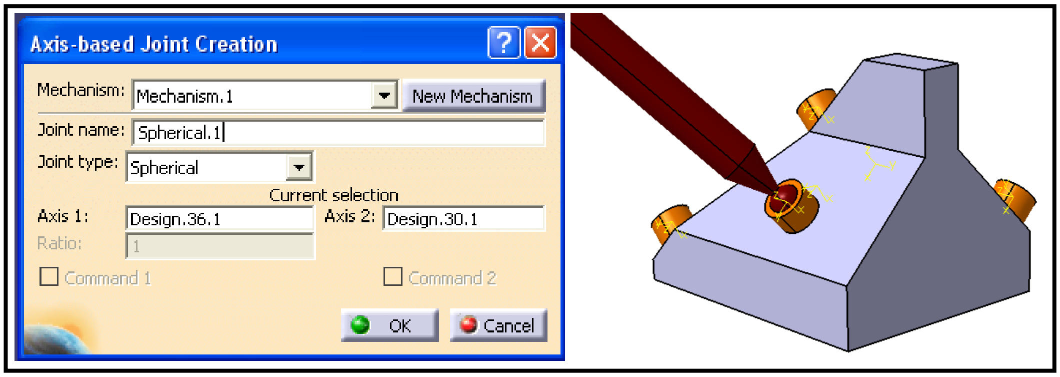


Figure E.7. FOIs for spherical joints

- Next, create the spherical joints by clicking *Joint From Axis* and selecting *Spherical* as joint type. The result should look like as



Axis-based Joint Creation

Mechanism: Mechanism.1

Joint name: Spherical.1

Joint type: Spherical

Current selection

Axis 1: Design.36.1

Axis 2: Design.30.1

Ratio: 1

□ Command 1

□ Command 2

Figure E.8. Spherical joint creation

- Then, repeat this action for the remaining links and the result should be as

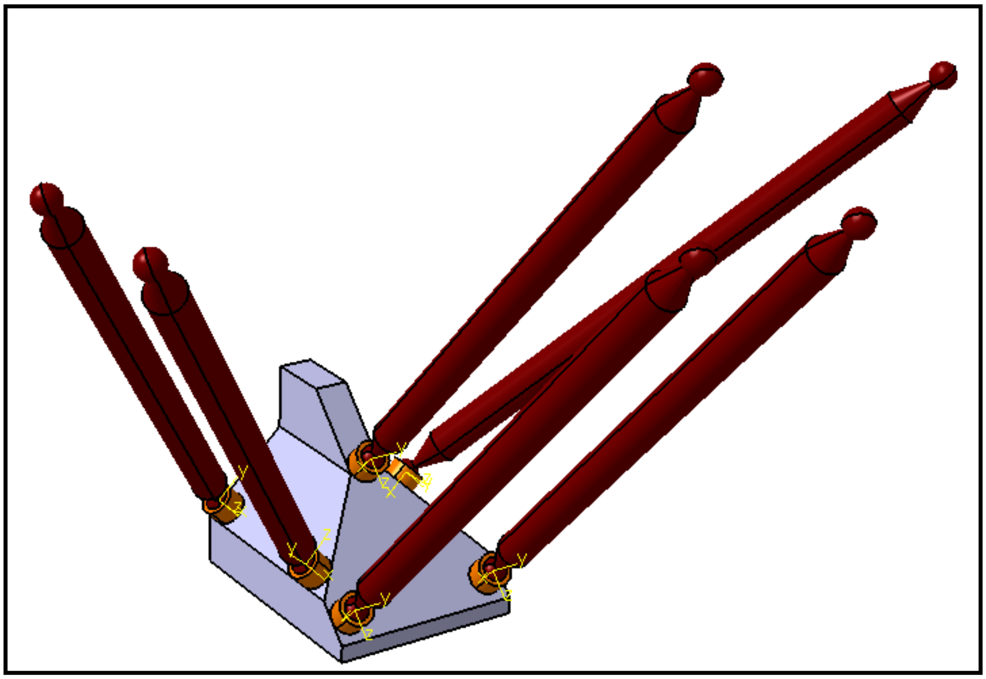


Figure E.9. Spherical joints for all legs

- Now, create the FOIs for the successive rotations' first revolute joints at upper attachment points. Initially, start creating the respective FOIs at the center of sockets in mobile platforms. The result should look like

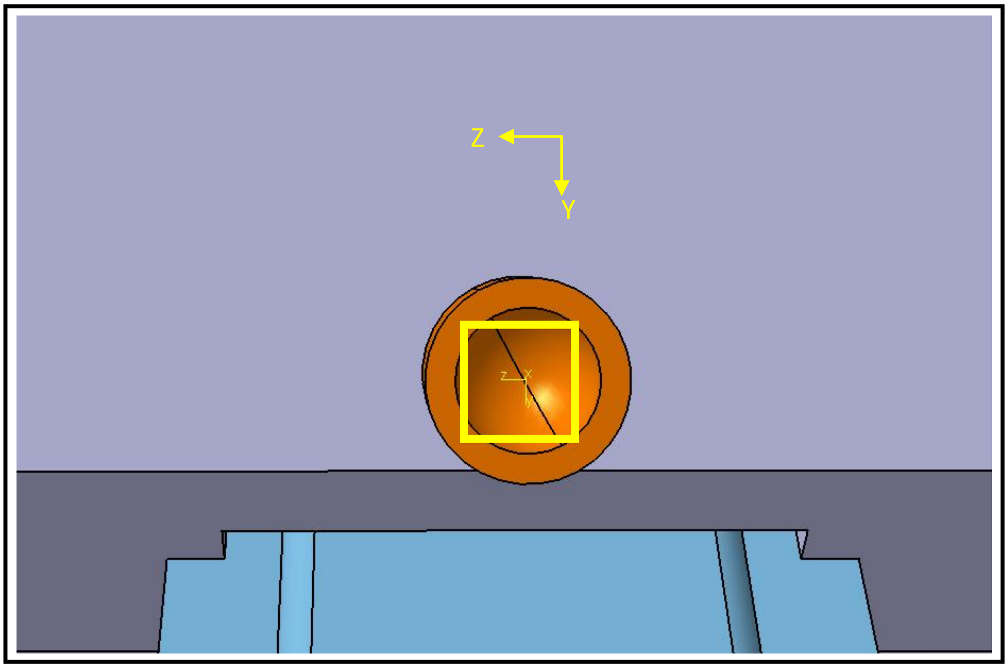


Figure E.10. FOIs for universal joint

- Now, go to pseudo part *ins_sph* and create a FOI folder. Inside that folder create a FOI and place it on the previous FOI at the same direction. Then, click *Joint From Axis* and create the revolute joint. The result should look like

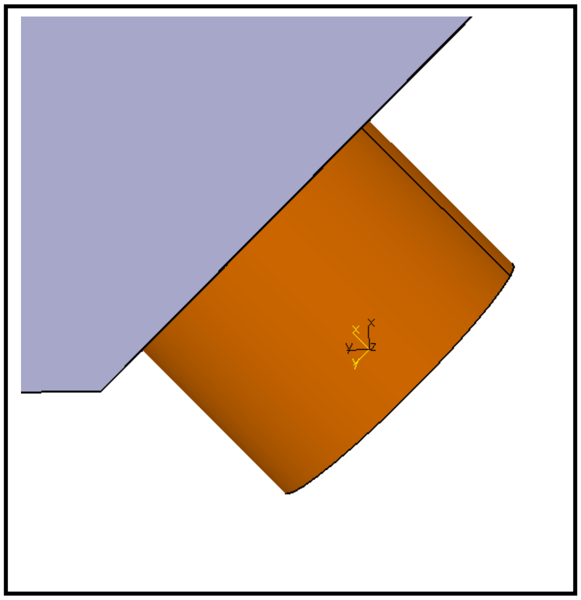


Figure E.11. First revolute joint of universal joint

- At the same location under the pseudo part, create another FOI with Z-direction matching previous FOI's Y-direction. The result should look like

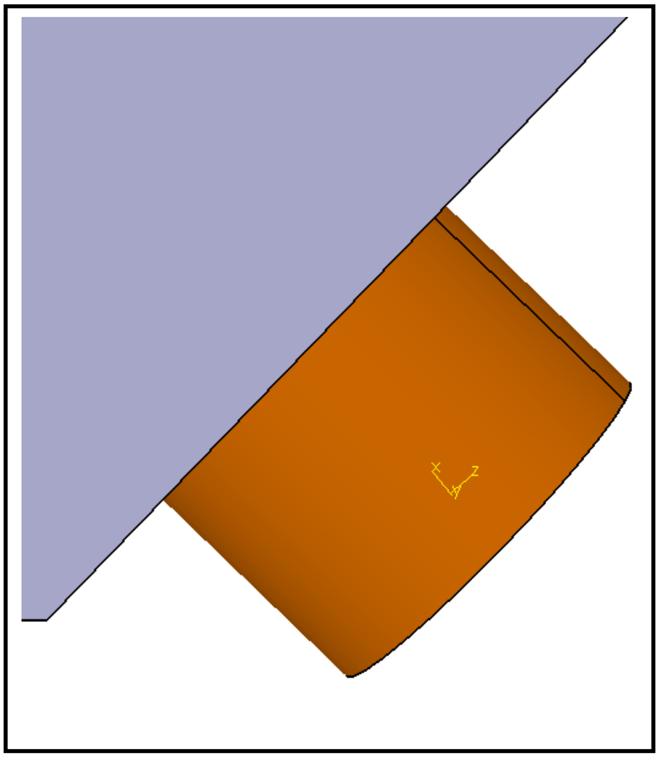


Figure E.12. Second set of FOIs for the second revolute joint of universal joint

- The corresponding FOI for previous frame will be created in the link's end that is not connected to the mobile platform. The orientation of this FOI does not matter since the other end is a spherical joint. Thus, the FOI in the link should be at the center of the ball and look like

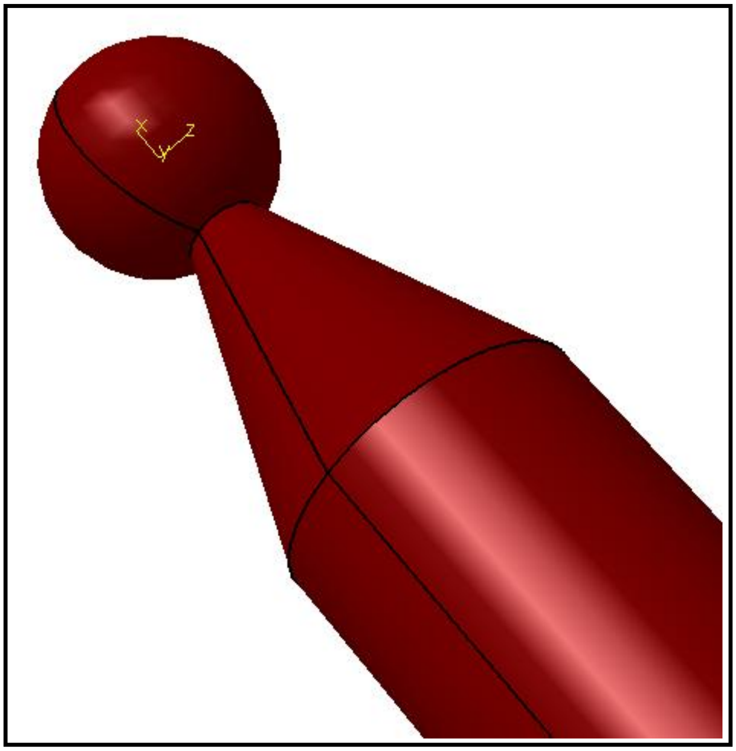


Figure E.13. FOI in the leg for the universal joint

- Now, create the second revolute joint between the respective FOIs. The result should look like

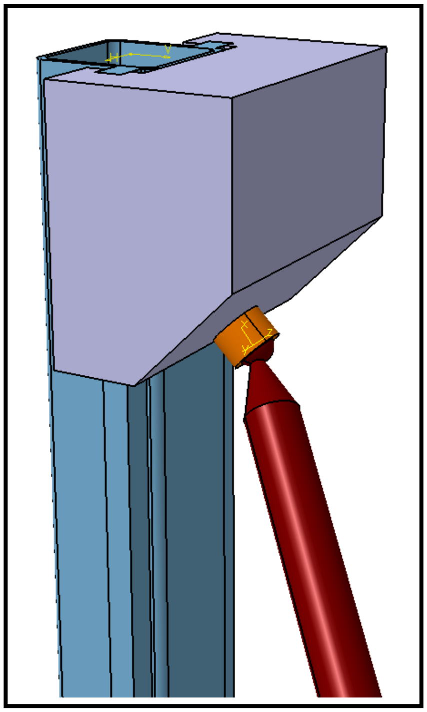


Figure E.14. Completed universal joint creation

- Then, repeat the same pattern for the remaining links and the result should look like

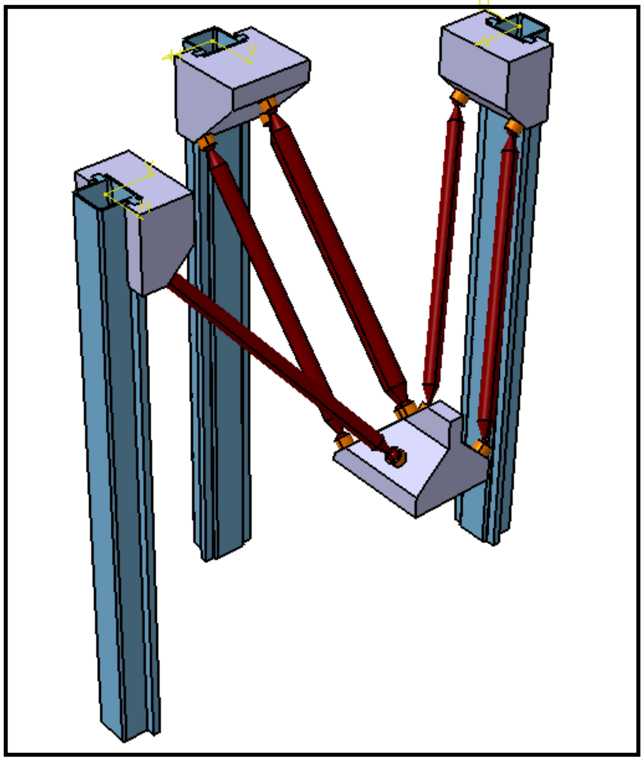


Figure E.15. Completed mechanism

- The last part of the forward kinematics is to place the pseudo part *Base_ref* to the center of the prismatic joint 1 (the cluster with 2 links). The result should look like

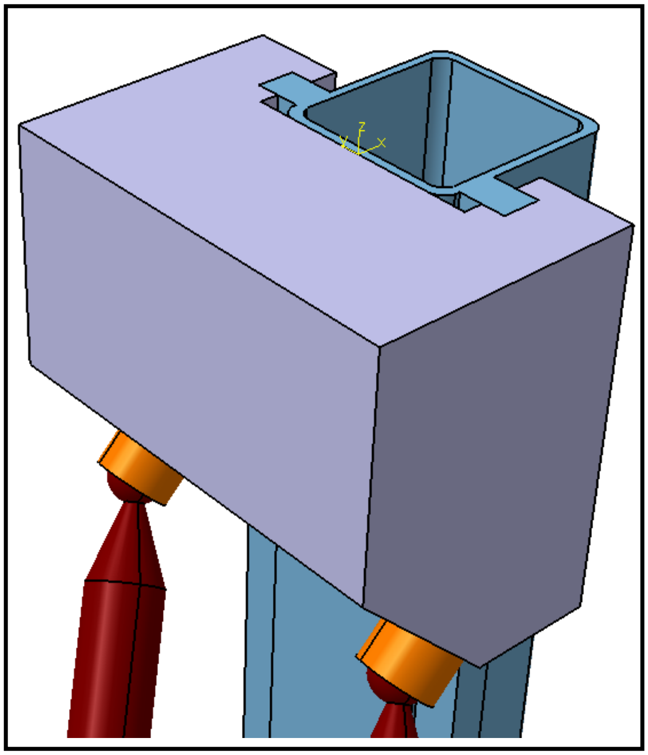


Figure E.16. FOI for the base reference

- In this step, click on the prismatic joints to make sure that their directions are in negative Z-direction of the base frame. The result should look like

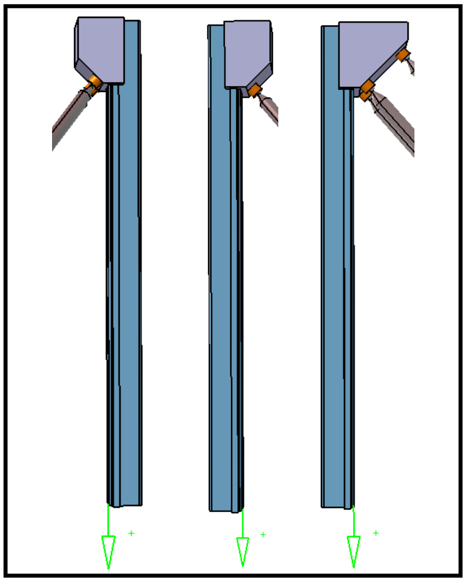


Figure E.17. Joint check

- In this step, appoint the travel limits by clicking button. The limits for all prismatic joints are [-1,2600] mm.
- Now, click on Jog Mechanism button to see that the behavior of the mechanism is correct.

INVERSE KINEMATICS: GANTRY-TAU

- First, create the TCP part and place it under the mobile platform at the center of the surface. The result should look like

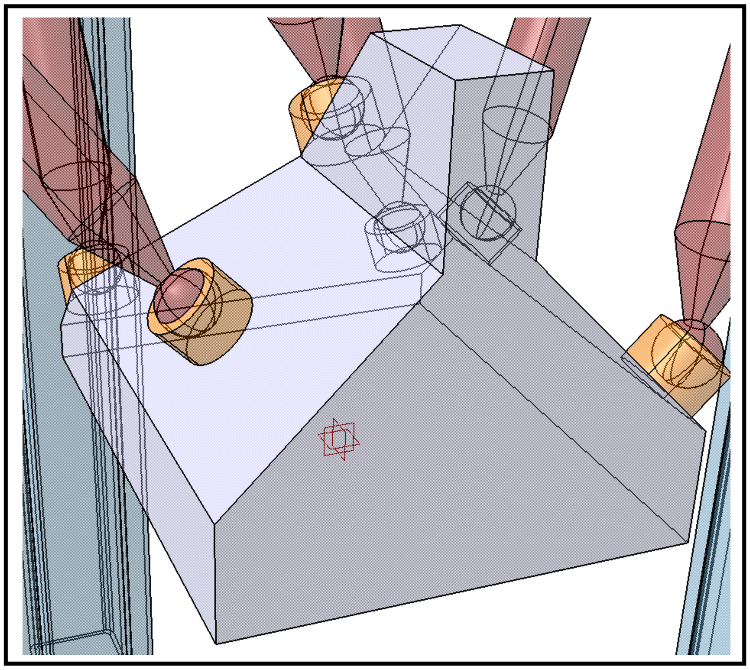


Figure E.18. Attaching new part to TCP

- Now, create the FOI for the mobile part at the center of TCP with the same orientation as base coordinate frame. The result should be as

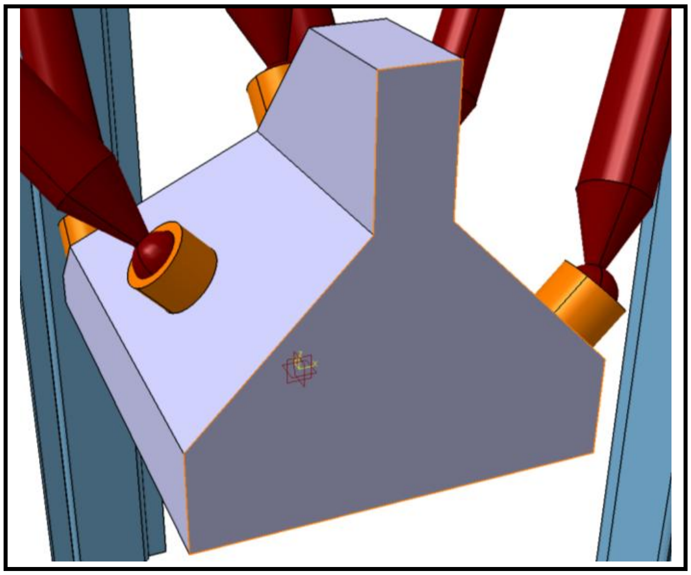
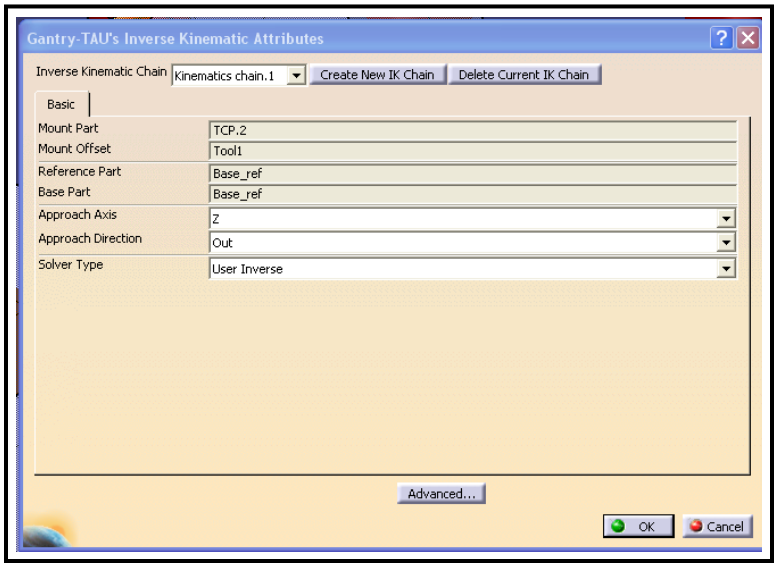


Figure E.19. Attaching tool FOI to TCP

- Then, click on Inverse Kinematics button and define *Basic* tab parameters as
    - **Mount Part:** TCP
    - **Mount Offset:** The FOI at the center of the wrist *Tool1*
    - **Base:** Pseudo part *Base_ref*
    - **Base reference:** Pseudo part *Base_ref*
    - **Approach Axis:** Z
    - **Approach Direction:** Out
    - **Solver Type:** User Inverse



Gantry-TAU's Inverse Kinematic Attributes

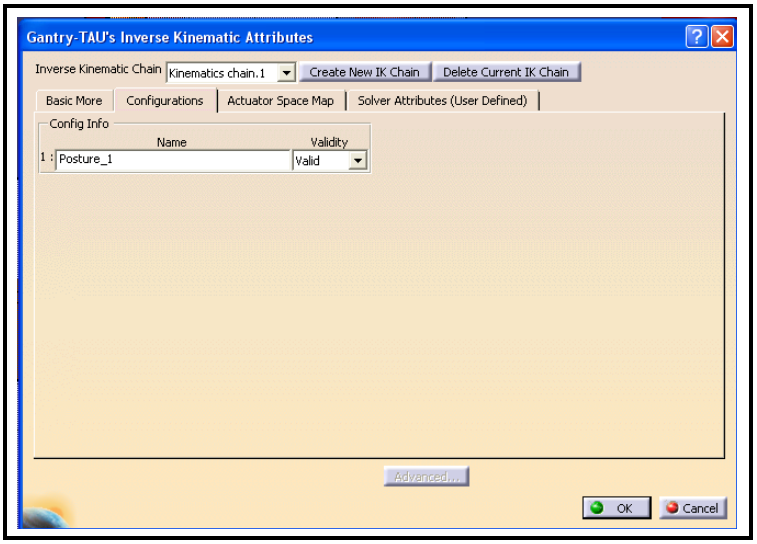
Inverse Kinematic Chain: Kinematics chain.1

Basic

Mount Part	TCP.2
Mount Offset	Tool1
Reference Part	Base_ref
Base Part	Base_ref
Approach Axis	Z
Approach Direction	Out
Solver Type	User Inverse

Figure E.20. Defining parameters for basic tab

- Now, click *Advanced* and make sure that *Posture_1* in *Configurations* tab is valid. The result should be as



Inverse Kinematic Chain: Kinematics chain.1

Config Info

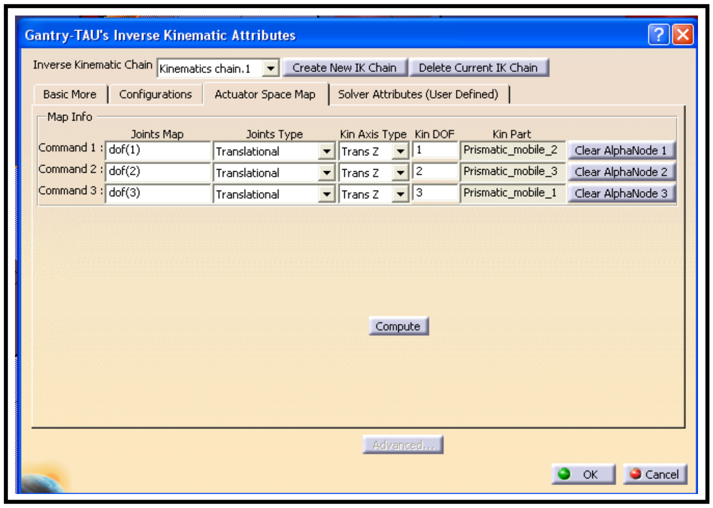
	Name	Validity
1 :	Posture_1	Valid

Figure E.21. Validating Posture for configurations tab

- In *Actuator Space Map* tab, the parameters should be as

	Joints Map	Joints Type	Kin Axis Type	Kin DOF	Kin Part
Command 1	dof(1)	Translational	Trans Z	1	*Prismatic_mobile_2*
Command 2	dof(2)	Translational	Trans Z	2	*Prismatic_mobile_3*
Command 3	dof(3)	Translational	Trans Z	3	*Prismatic_mobile_1*

- The result in the tab should look like



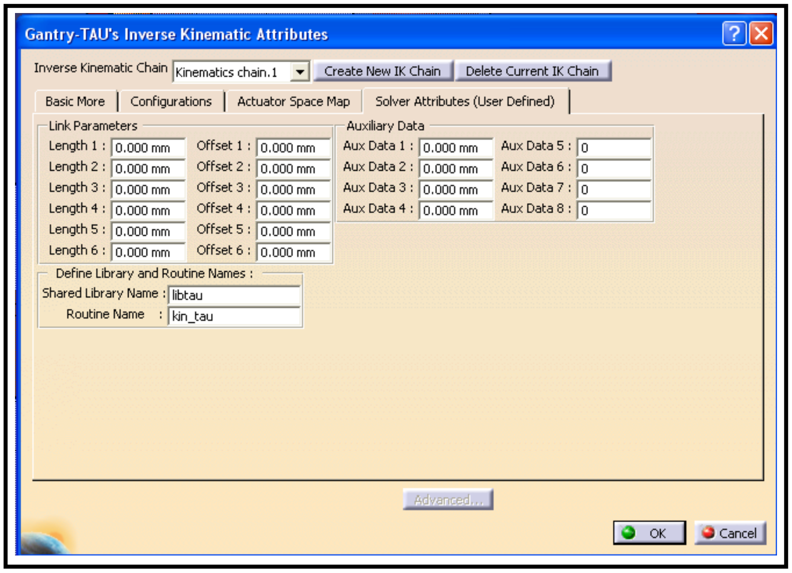
Inverse Kinematic Chain: Kinematics chain.1

Map Info

	Joints Map	Joints Type	Kin Axis Type	Kin DOF	Kin Part
Command 1 :	dof(1)	Translational	Trans Z	1	Prismatic_mobile_2
Command 2 :	dof(2)	Translational	Trans Z	2	Prismatic_mobile_3
Command 3 :	dof(3)	Translational	Trans Z	3	Prismatic_mobile_1

Figure E.22. Defining parameters for actuators

- In the last tab – *Solver Attributes*, fill the library name as *libtau* and the routine name as *kin_tau*. The tab should be as



Inverse Kinematic Chain: Kinematics chain.1

Link Parameters

Length 1 :	0.000 mm	Offset 1 :	0.000 mm
Length 2 :	0.000 mm	Offset 2 :	0.000 mm
Length 3 :	0.000 mm	Offset 3 :	0.000 mm
Length 4 :	0.000 mm	Offset 4 :	0.000 mm
Length 5 :	0.000 mm	Offset 5 :	0.000 mm
Length 6 :	0.000 mm	Offset 6 :	0.000 mm

Auxiliary Data

Aux Data 1 :	0.000 mm	Aux Data 5 :	0
Aux Data 2 :	0.000 mm	Aux Data 6 :	0
Aux Data 3 :	0.000 mm	Aux Data 7 :	0
Aux Data 4 :	0.000 mm	Aux Data 8 :	0

Define Library and Routine Names :

Shared Library Name : libtau

Routine Name : kin_tau

Figure E.23. Defining library and C-file names

- Then, click *OK* and start testing the robot by using the *Jog Mechanism* button as in earlier cases and check whether the inverse kinematics' joint values are matching the actual values by on the *Mechanism* tab.

FORWARD KINEMATICS THEORY OF GANTRY-TAU ROBOT

The theory of forward kinematics of Gantry-Tau robot will be presented in this section in order to prove that DELMIA V5's posture change for this robot is resulting in two different TCP coordinates. Also, not to interrupt the consistency of the structure of the robot it was decided to present the relative theory as appendix.

As Johannesson (2003) states the forward kinematics of Gantry-Tau robot can be presented as three spheres created for each cluster; thus, the intersection(s) of these spheres will yield the TCP coordinates.

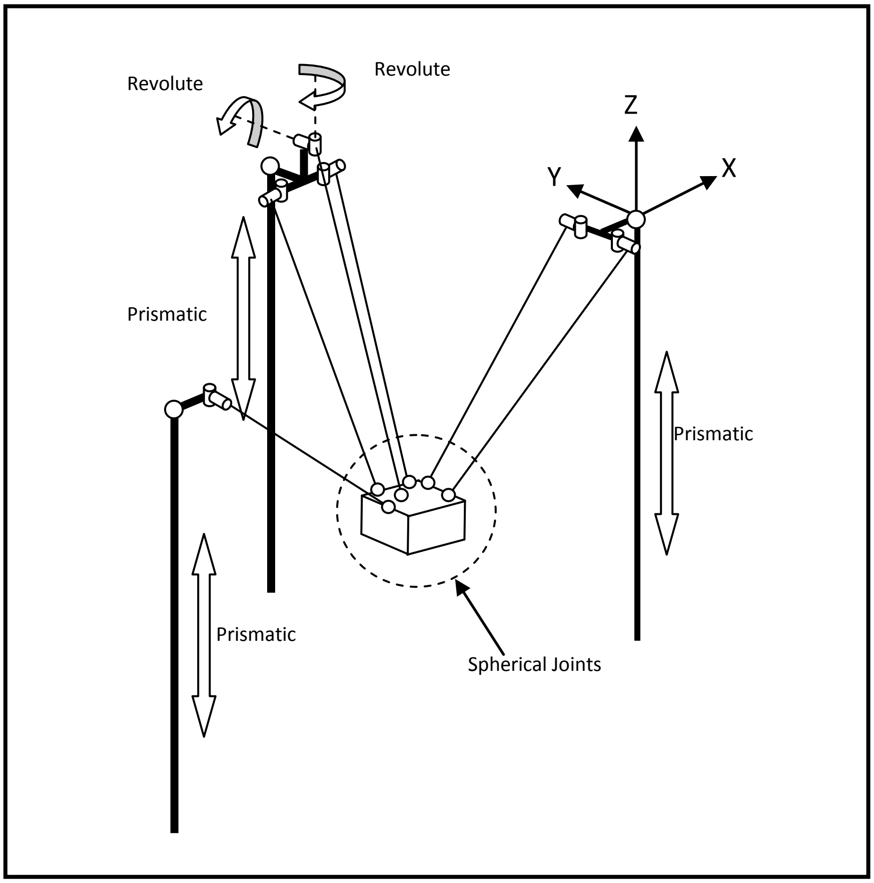


Figure E.24. Gantry-TAU schematics

If, again, the vectors for each cluster are formulated with respect to TCP values, the result then

$$\vec{P_i} + \vec{d_i} + \vec{l_i} = \vec{T} + \vec{n_i} \qquad (i = 1,2,3)$$

where $\vec{P_i}$ is the vector that defines prismatic actuators, $\vec{d_i}$ is the vector from the end of prismatic joint vector the universal joints. $\vec{n_i}$ is the vector that connects TCP ($\vec{T}$) to the attachment points of legs on the mobile platform. Thus, when this equation is written with components and its absolute value is taken for lengths of the each cluster, the result will be

$$l_i{}^2 = (T_{i,X} + n_{i,X} - P_{i,X} - \mathrm{d}_{\mathrm{i,X}})^2 + (T_{i,Y} + n_{i,Y} - P_{i,Y} - \mathrm{d}_{\mathrm{i,Y}})^2 + (T_{i,Z} + n_{i,Z} - P_{i,Z} - \mathrm{d}_{\mathrm{i,Z}})^2$$

Thus, the spherical equation above can be easily solved and since the equation is a second order polynomial, the result of this equation will yield two sets of solutions for $\vec{T}$ vector; hence, TCP coordinates get two different values for a set of given joint values. The MATLAB function of this calculation can be seen in appendix G.

THE C-FILE FOR GANTRY-TAU ROBOT

```
/****************************************************************************
***
**
**                      USER KINEMATICS EXAMPLE
**
**  Copyright (c) 1990 Delmia Corporation, All rights reserved.
**
**  This file contains an example of a kinematics routine for the
**  shared library.  This example will work for 4 DOF 2 Config (left and
right
**  elbow) scara robots such as the ASEA/IRB300.  By default,
**  kin_usr1 is mapped to this routine.
**
**  For a description of kinematics solutions refer to:
**
**      Paul, Richard P., "Robot Manipulators: Mathematics, Programming
**      and Control", The MIT Press, Cambridge, Massachusetts, 1981.
**
**  DESCRIPTION OF ARGUMENTS
**
**  double T6[4][4]        4x4 position matrix of center of wrist. This is
**                         the goal point MINUS the tool frame and mounting
**                         plate offsets. This is the easiest point to start
**                         the inverse kinematic solution from, and is the
**                         traditional approach.
**
**                         NOTE: T6 matrix may be transposed from your usual
**                         notation.
**
**                              | nx ny nz 0 | \\
**                         T6 = | ox oy oz 0 |  > direction cosines (9)
**                              | ax ay az 0 | /
**                              | px py pz 1 | -> position terms (3)
**
**                         px = T6[3][0];
**
**  double link_lengths[]  Distance between joint axis along link length
**
**  double link_offsets[]  Offset between joint axis along joint axis
**
**                         These two arrays can be considered the Denevitt-
**                         Hartenburg variables described in Paul's book, or
**                         any convenient scheme the user desires.
**
**  double solutions[][]   A two dimensional array contains all possible
**                         solutions for robot arm. It is up to user to
**                         decide how many solutions are possible, and to
**                         provide all solutions when routine is called:
**                         elbow up, elbow down, etc.  The CONFIGS
**                         Button in IGRIP allows user to view all possible
**                         solutions and may provide insight into importance
**                         of this array.
**
**  int warnings[]         Array providing warning states for each solution
**                         such as unreachable, singular, etc. Possible
warning
**                         states are defined in include file shlibdefs.h
**                         and are:
**
**                         WARN_GOOD_SOLUTION
**                         WARN_JOINT_LIMIT_EXCEEDED
**                         WARN_UNREACHABLE
**                         WARN_SINGULAR_SOLUTION
**
**       NOTE: shlibdefs.h is automatically included by the IGRIP Shared
**             Library Make system.  For further details regarding the
building
**             of the shared library, refer to the IGRIP Motion Pipeline
**             Reference Guide
**
**
**  Words of encouragement
**
**      Writing inverse kinematics routines is a challenge. Invariably
**      you will make mistakes which later seem trivial.  Even experts on
**      the subject loathe writing a new routine.  The usual problems
**      are matching the routines view of the world with the device
**      definition.  You must check that where this routine thinks is
**      the axis origin, or the zero reference position, is the same
**      as the IGRIP device.  Also make sure that each agree upon the
positive
**      sense of direction.  These are the most common foul ups.  Next,
**      the mounting plate offset may be wrong, so when first debugging
**      your routine, set the mounting plate and tool frame offsets to
**      zero.  Next check for dropped signs in your equations.  Maybe
**      an inverse trig function is returning an angle in a different
quadrant
**      than the one you want.  Perhaps you should be using atan2 instead
**      of atan (or vice-versa).  Remember that trig and inverse trig
function
**      angles are in radians.  Also, check array indices.  Remember that
**      arrays start at zero not one, so link_4's offset is at
link_offsets[3].
**      Are you referring to T6[3][2], when you mean T6[2][3]? Remember that
**      transformation matrices may be transposed from standard text book
**      definitions.  Once you get your routine to work you will have earned
**      the title of kinematician.
**
*****************************************************************************
**/

#include <shlibdefs.h>

/*
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
```

```
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
**
** USER SHOULD CHANGE THESE VALUES APPROPRIATELY
**                       |
**                       |
**                      \ /
**                       v                                         */

#define NUM_SOLUTIONS    8        /* Number of possible solutions  */
#define NUM_DOFS         3        /* Number of joints to be solved */

/*                       ^
**                      / \
**                       |
**                       |
** USER SHOULD CHANGE THESE VALUES APPROPRIATELY
**
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
** IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT  IMPORTANT
IMPORTANT
*/
/*
 * User must supply this function
*/

DllExport int
get_kin_config( char *kin_routine, int *kin_dof, int *solution_count, int
*usrKinDataHint )
{
      if( strcmp( kin_routine, "kin_tau" ) == 0 )
      {
            *kin_dof = NUM_DOFS;
            *solution_count = NUM_SOLUTIONS;
                  /*
                  * this indicates kin_usr's last argument (void *pData)
                  * will be DLM_Data_KinStat
                  */
                  *usrKinDataHint = USR_KIN_DATA_KINSTAT;
                  return 0;
      }
      return 1;
}
static char JointType[2][24] = { "ROTATIONAL", "TRANSLATIONAL" };
static char KinMode[2][24] = { "Normal", "TrackTCP" };
```

```
/*
** Routine Name
*/
DllExport int
kin_tau(
      link_lengths,
      link_offsets,
      T6, /* See above for description of these arguments */
      solutions,
      warnings,
            pData
      )


/*
** Passed Variable Declarations
*/
double T6[4][4],
      link_lengths[],
      link_offsets[],
      solutions[][NUM_SOLUTIONS];
int warnings[];

void *pData; /* usr routine should NEVER delete pData */
{
/*
** Local Variable Declarations (add variable declarations as appropriate)
*/
long double nx, ny, nz, ox, oy, oz, ax, ay, az, px, py, pz;
long double D11, D12, D13, D21, D22, D23, D31, D32, D33;
long double J11,J12,J21,J22,J31,J32, Lref,Lref2;
//Variables to perform matrix multiplication
int row1,row2,row3;
int col1,col2,col3;
int inner1,inner2,inner3;
// The prismatic joint vectors for each leg (The vector between the TCP and
each upper attachment point).
long double P1[4][1];
long double P2[4][1];
long double P3[4][1];
// The vectors that connect upper attachment points to prismatic joints
long double d1[4][1];
long double d2[4][1];
long double d3[4][1];
// The vectors that connect lower attachment points to TCP
long double n1[4][1];
long double n2[4][1];
long double n3[4][1];
// The resulting vectors of multiplication of ni vectors with TCP matrix
long double NT1[4][1] = {0};
long double NT2[4][1] = {0};
long double NT3[4][1] = {0};
//The transformed T6 matrix named as TCP matrix (do not confuse with Tool
Centre Point) - see line 307.
long double TCP[4][4];
```

```
#if 1
/*
* using pData
*/
      int i;

      DLM_Data_KinStat *pDLM_Data = (DLM_Data_KinStat *) pData;
      if( pDLM_Data )
      {
            printf( "\n\ndof_count: %d\n", pDLM_Data->dof_count );

            printf( "\njoint_types:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%s ", JointType[(pDLM_Data->joint_types)[i]] );

            printf( "\n\nkin_mode: %s\n", KinMode[pDLM_Data->kin_mode] );

            printf( "\njoint_values:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->joint_values[i] );

            printf( "\n\njnt_trvl_lmts lower:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[0][i] );

            printf( "\n\njnt_trvl_lmts upper:\n" );
            for( i = 0; i < pDLM_Data->dof_count; i++ )
                  printf( "%12.4f ", pDLM_Data->jnt_trvl_lmts[1][i] );

            printf( "\n\n" );

      }
#endif


/***--------------- Execution Begins Here ----------------------------------
***/
      /*
      ** DO NOT REMOVE THIS BLOCK OF CODE
      ** IT IS REQUIRED TO PROPERLY SET THE NUMBER OF KINEMATIC
      ** DOFS FOR THE DEVICE
      */
      if( !kin_check_definition( NUM_DOFS, NUM_SOLUTIONS ) )
      {
      /*
      ** Inconsistency between device definition and inverse
      ** kinematics routine exists. A warning message has been
      ** issued and routine aborted
      */
      return( 1 );
      }
/***---------------- User code begins here ---------------------------------
***/
```

```
//The vectors to define the prismatic joints where Pi[2][0] is the prismatic
joint value in negative direction
P1[0][0] =      0;  P1[1][0] =       0; P1[2][0] = 0; P1[3][0] = 1;
P2[0][0] =   -1100; P2[1][0] =    700; P2[2][0] = 0; P2[3][0] = 1;
P3[0][0] =   -2200; P3[1][0] =      0; P3[2][0] = 0; P3[3][0] = 1;

//The constant vectors to define the upper attachment points from prismatic
joint end
d1[0][0] =   -96.569; d1[1][0] =     -185; d1[2][0] =     -400; d1[3][0] = 1;
d2[0][0] =         0; d2[1][0] = -322.843; d2[2][0] = -173.726; d2[3][0] = 1;
d3[0][0] =    96.569; d3[1][0] =        0; d3[2][0] =     -400; d3[3][0] = 1;

//The vectors that connect TCP to lower attachment points.
n1[0][0] =  224.999; n1[1][0] =  -240.001; n1[2][0] =  171.568; n1[3][0] = 1;
n2[0][0] =        0; n2[1][0] =    39.705; n2[2][0] =  336.862; n2[3][0] = 1;
n3[0][0] = -182.574; n3[1][0] =       -80; n3[2][0] =  213.994; n3[3][0] = 1;

//Reference lengths
      Lref=1500 ;
      Lref2=1499.775;

//Importing the current TCP values from Delmia through the T6 matrix and
putting proper context
      nx = T6[0][0];
      ny = T6[0][1];
      nz = T6[0][2];
      ox = T6[1][0];
      oy = T6[1][1];
      oz = T6[1][2];
      ax = T6[2][0];
      ay = T6[2][1];
      az = T6[2][2];
      px = T6[3][0];
      py = T6[3][1];
      pz = T6[3][2];

//The transforming T6 matrix from row vectors form to column vector form
TCP[0][0] =  nx; TCP[0][1] =  ox; TCP[0][2] =  ax; TCP[0][3] = px;
TCP[1][0] =  ny; TCP[1][1] =  oy; TCP[1][2] =  ay; TCP[1][3] = py;
TCP[2][0] =  nz; TCP[2][1] =  oz; TCP[2][2] =  az; TCP[2][3] = pz;
TCP[3][0] =  0;  TCP[3][1] = 0;   TCP[3][2] =  0;  TCP[3][3] = 1;

//Calculating the current position (in x,y,z in Base coordinates) of each
lower attachment point for each leg by multiplying the transformation
// matrix TCP[4][4] with the vector between the current TCP (the T6 matrix)
and the lower attachmentpoint for each leg

//Calculate upper position on Leg1 (The array L1tCur)
 for (row1 = 0; row1 < 4; row1++) {
        for (col1 = 0; col1 < 1; col1++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner1 = 0; inner1 < 4; inner1++) {
                NT1[row1][0] += TCP[row1][inner1] * n1[inner1][col1];
            }
        }
  }
 //Calculate upper position on Leg2 (The array L2tCur)
    for (row2 = 0; row2 < 4; row2++) {
        for (col2 = 0; col2 < 1; col2++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner2 = 0; inner2 < 4; inner2++) {
                NT2[row2][0] += TCP[row2][inner2] * n2[inner2][col2];
            }
        }
  }
//Calculate upper position on Leg3 (The array L3tCur)
    for (row3 = 0; row3 < 4; row3++) {
        for (col3 = 0; col3 < 1; col3++) {
            // Multiply the row of A by the column of B to get the row,
column of product.
            for (inner3 = 0; inner3 < 4; inner3++) {
                NT3[row3][0] += TCP[row3][inner3] * n3[inner3][col3];
            }
        }
  }
//Finding the joint values by using the theory. Multiplication with -1 stems
from the direction of the joints.

J11=-1*(NT1[2][0]-d1[2][0]+ sqrt(pow(Lref,2)-pow((P1[0][0]+d1[0][0]-
NT1[0][0]),2)-pow((P1[1][0]+d1[1][0]-NT1[1][0]),2)));

J12=-1*(NT1[2][0]-d1[2][0]- sqrt(pow(Lref,2)-pow((P1[0][0]+d1[0][0]-
NT1[0][0]),2)-pow((P1[1][0]+d1[1][0]-NT1[1][0]),2)));

J21=-1*(NT2[2][0]-d2[2][0]+ sqrt(pow(Lref,2)-pow((P2[0][0]+d2[0][0]-
NT2[0][0]),2)-pow((P2[1][0]+d2[1][0]-NT2[1][0]),2)));

J22=-1*(NT2[2][0]-d2[2][0]- sqrt(pow(Lref,2)-pow((P2[0][0]+d2[0][0]-
NT2[0][0]),2)-pow((P2[1][0]+d2[1][0]-NT2[1][0]),2)));
J31=-1*(NT3[2][0]-d3[2][0]+ sqrt(pow(Lref2,2)-pow((P3[0][0]+d3[0][0]-
NT3[0][0]),2)-pow((P3[1][0]+d3[1][0]-NT3[1][0]),2)));

J32=-1*(NT3[2][0]-d3[2][0]- sqrt(pow(Lref2,2)-pow((P3[0][0]+d3[0][0]-
NT3[0][0]),2)-pow((P3[1][0]+d3[1][0]-NT3[1][0]),2)));

//Sending the final joint values back to the "solutions"-matrix which is the
input matrix for Delmia.
solutions[0][0] = J11;  solutions[1][0] = J21; solutions[2][0] = J31;
solutions[0][1] = J11;  solutions[1][1] = J21; solutions[2][1] = J32;
solutions[0][2] = J11;  solutions[1][2] = J22; solutions[2][2] = J31;
solutions[0][3] = J11;  solutions[1][3] = J22; solutions[2][3] = J32;
solutions[0][4] = J12;  solutions[1][4] = J21; solutions[2][4] = J31;
solutions[0][5] = J12;  solutions[1][5] = J21; solutions[2][5] = J32;
solutions[0][6] = J12;  solutions[1][6] = J22; solutions[2][6] = J31;
solutions[0][7] = J12;  solutions[1][7] = J22; solutions[2][7] = J32;
```

```
//Printing some of the variable values out in the debug window to ease
debugging and get an overview of what is going on
//Printing the current TCP values in the debug window for evaluation purposes
printf( "\nx ny nz: %12.4f ,%12.4f ,%12.4f\n", nx ,ny ,nz );
printf( "\ox oy oz: %12.4f ,%12.4f ,%12.4f\n", ox ,oy ,oz );
printf( "\ax ay az: %12.4f ,%12.4f ,%12.4f\n", ax ,ay ,az );
printf( "\px py pz: %12.4f ,%12.4f ,%12.4f\n", px ,py ,pz );

//Printing the transformed matrix TCP
printf( "\n Transformed T6 matrix - TCP matrix\n");
printf( "\nx ox ax px: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", nx ,ox ,ax, px );
printf( "\ny oy ay py: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", ny ,oy ,ay, py );
printf( "\nz oz az pz: %12.4f ,%12.4f ,%12.4f ,%12.4f\n", nz ,oz ,az, pz );

//Joint values
printf( "\n The joint values\n" );
printf( "J11 J21 J31: %12.4f ,%12.4f ,%12.4f\n", J11 ,J21 ,J31 );
printf( "J11 J21 J32: %12.4f ,%12.4f ,%12.4f\n", J11 ,J21 ,J32 );
printf( "J11 J22 J31: %12.4f ,%12.4f ,%12.4f\n", J11 ,J22 ,J31 );
printf( "J11 J22 J32: %12.4f ,%12.4f ,%12.4f\n", J11 ,J22 ,J32 );
printf( "J12 J21 J31: %12.4f ,%12.4f ,%12.4f\n", J12 ,J21 ,J31 );
printf( "J12 J21 J32: %12.4f ,%12.4f ,%12.4f\n", J12 ,J21 ,J32 );
printf( "J12 J22 J31: %12.4f ,%12.4f ,%12.4f\n", J12 ,J22 ,J31 );
printf( "J12 J22 J32: %12.4f ,%12.4f ,%12.4f\n", J12 ,J22 ,J32 );


warnings[ 0 ] = WARN_GOOD_SOLUTION;


return (0);
}
```

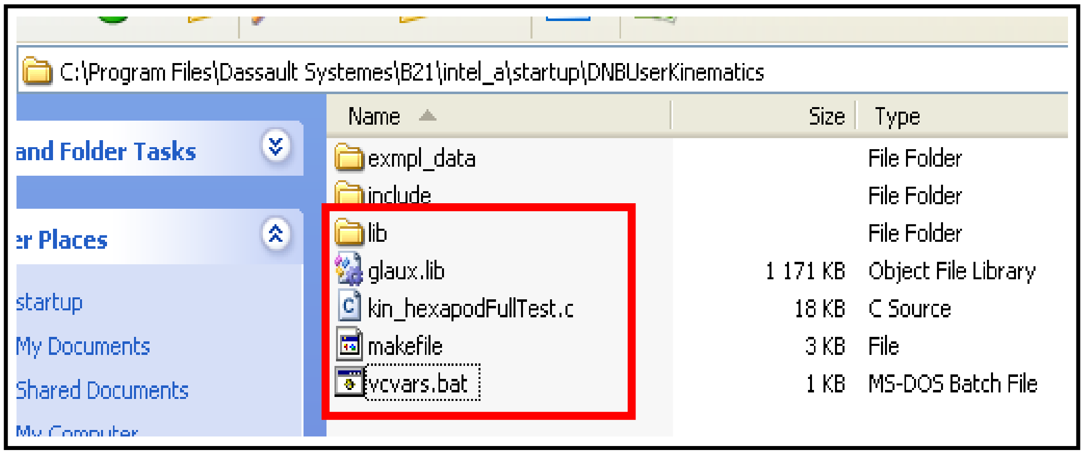
APPENDIX F: COMPILATION OF C-FILES

The compilation of C-files and their placement in DELMIA V5 folder will be covered in this section. Before this operation being explained, there is a prerequisite that users must fulfill – which is a compilation tool.

Dassault Systemes endorses the use of Microsoft Visual Studio (version 8 or higher). On the other hand, it would be possible to use other compilation tools which support the use of *nmake all* command. It must also be noted that the compilation tool must support C# language since the basic version of Visual Studio does not support the compilation of C-files.

COMPILATION IN 64-BIT OPERATING SYSTEMS

- Go to "***C:\Program Files (x86)\Dassault Systemes\B21\win_b64\startup\DNBUserKinematics***". The directory may vary depending on the user.
- At this point make sure that the ***DNBUserKinematics*** folder has ***glaux.lib*** file. If not, copy and paste the file directly without making any change. This file ***glaux.lib*** is a standard library file and it can be directly found on various sources.
- Create a folder named ***lib*** under the same directory.
- Also copy and paste the ***vcvars.bat*** file there.
- Copy and paste the C-file created in this directory as well. The result should look like this

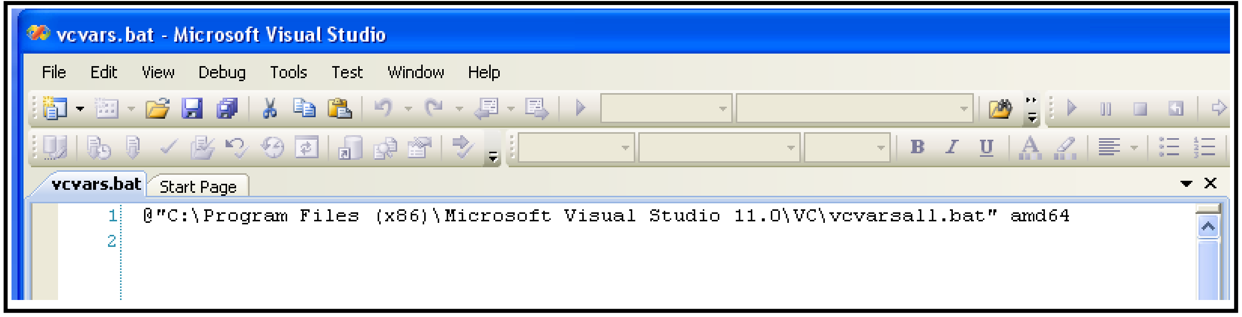


C:\Program Files\Dassault Systemes\B21\intel_a\startup\DNBUserKinematics

Name	Size	Type
exmpl_data		File Folder
include		File Folder
lib		File Folder
glaux.lib	1 171 KB	Object File Library
kin_hexapodFullTest.c	18 KB	C Source
makefile	3 KB	File
vcvars.bat	1 KB	MS-DOS Batch File

Figure F.1. DNBUserKinematics folder

- Now, open ***vcvars.bat*** file with ***VS, WordPad*** or ***NotePad*** and it should look like



```
@"C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\vcvarsall.bat" amd64
```

Figure F.2. vcvars.bat file for compilation tool

- Now, change the directory for the version of Visual Studio used or any other compilation tool available. For example, in this thesis work VS 9 is used; thus, the line should be

```
@"C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"
amd64
```

- Save and close ***vcvars.bat*** file and proceed to the next step.
- At this point, necessary changes in ***makefile*** will be made. Open ***makefile*** in ***VS***, ***WordPad*** or ***NotePad.***
- In ***makefile***, go to lines 1-5 (which specify the names of library and locations that will be created after compilation) and find the following code and change as following:

```
#
# library name and locations
#
DEST                = .\lib
LIBRARY             = libhexapodFullTest
```

- Go to line 10 and paste directory of the Visual Studio. It should look like as

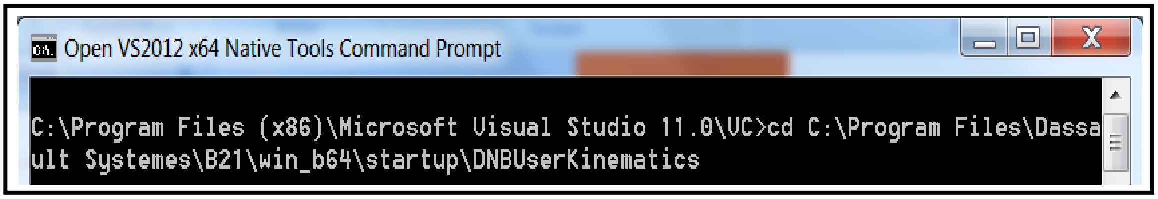
```
#
# set up the MS Visual C++ compiler
#
MS_LOC= c:\Program Files (x86)\Microsoft Visual Studio 9.0\VC
```

- **OBS!** For other versions of VS, change the directory accordingly.
- On line 58, the name of the C-file created before should be specified. Thus, it should look like as following (notice that it should end with *obj* extension)

```
#
# files to compile
#
OBJS                = \
               kin_hexapodFullTest.obj
```

- When the changes are made, save and close the ***makefile***.
- **OBS!** Other versions of DELMIA provide different ***makefile***s; however the changes in the file are still the same as above. The only difference is in the number of lines in which these changes should be made.
- Now, proceed to compilation operation. First, open the VS command prompt and enter the following command

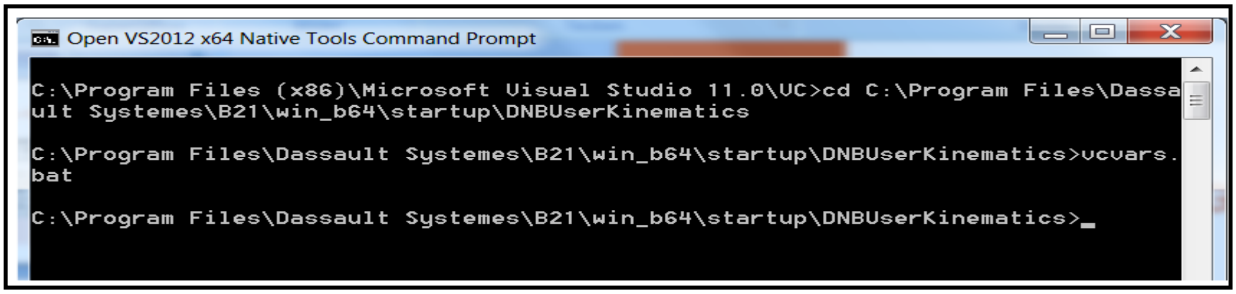
  cd C:\Program Files\Dassault Systemes\B21\win_b64\startup\DNBUserKinematics



```
C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC>cd C:\Program Files\Dassa
ult Systemes\B21\win_b64\startup\DNBUserKinematics
```

Figure F.3. VS command prompt with folder destination

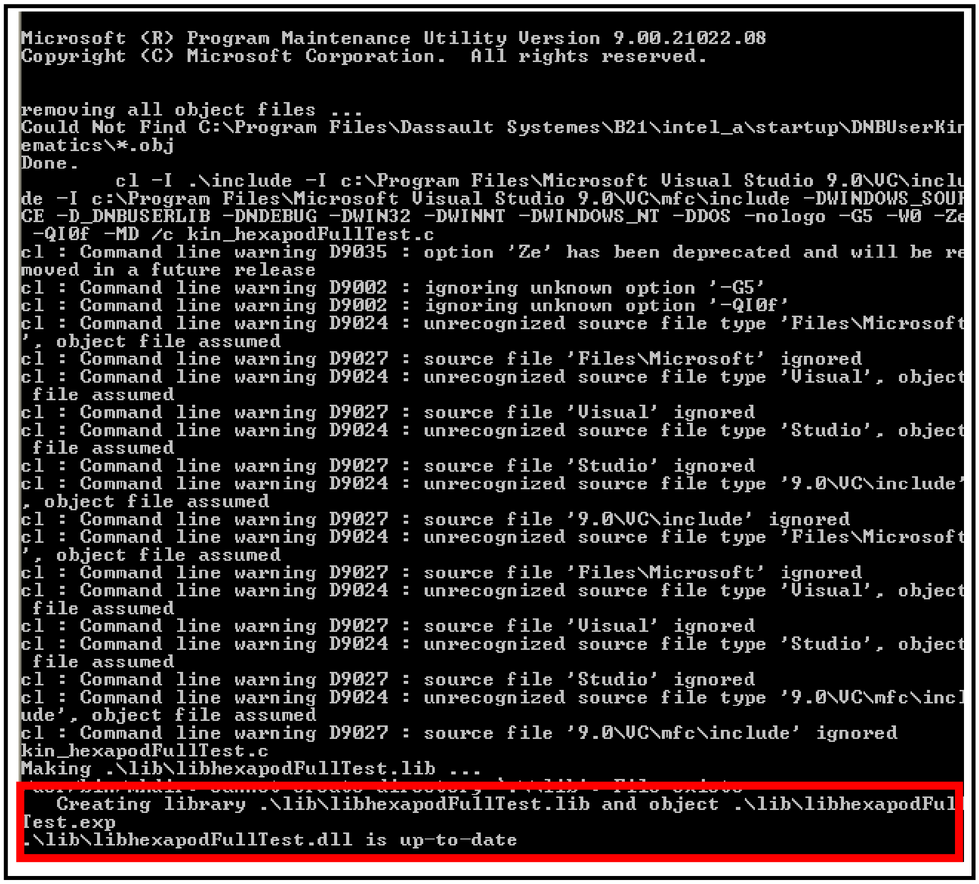
- Under the same directory, type ***vcvars.bat*** and press *enter.* The result should look like



```
C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC>cd C:\Program Files\Dassa
ult Systemes\B21\win_b64\startup\DNBUserKinematics

C:\Program Files\Dassault Systemes\B21\win_b64\startup\DNBUserKinematics>vcvars.
bat

C:\Program Files\Dassault Systemes\B21\win_b64\startup\DNBUserKinematics>_
```

Figure F.4. VS command prompt after executing vcvars.bat

- Now, the compilation operation can be done. Again, remain under the same directory in command prompt, and write ***nmake all.*** If C-file has no errors, the command prompt will look like



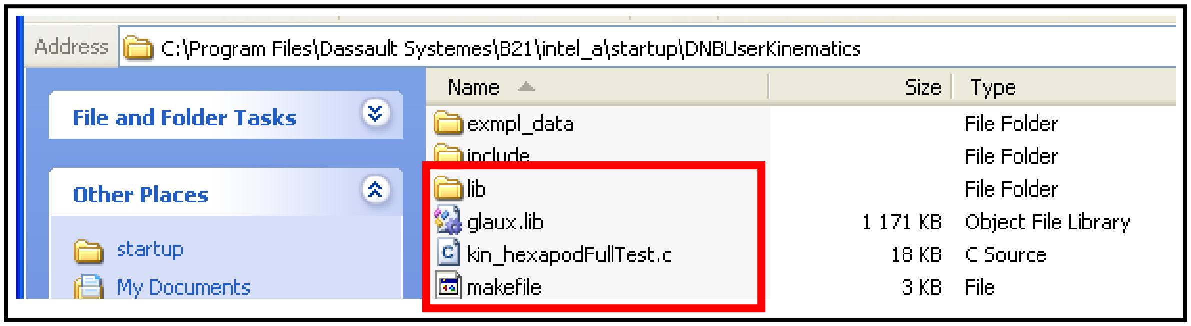
```
Microsoft (R) Program Maintenance Utility Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.


removing all object files ...
Could Not Find C:\Program Files\Dassault Systemes\B21\intel_a\startup\DNBUserKin
ematics\*.obj
Done.
        cl -I  .\include -I c:\Program Files\Microsoft Visual Studio 9.0\VC\inclu
de -I c:\Program Files\Microsoft Visual Studio 9.0\VC\mfc\include -DWINDOWS_SOUR
CE -D_DNBUSERLIB -DNDEBUG -DWIN32 -DWINNT -DWINDOWS_NT -DDOS -nologo -G5 -W0 -Ze
 -QI0f -MD /c kin_hexapodFullTest.c
cl : Command line warning D9035 : option 'Ze' has been deprecated and will be re
moved in a future release
cl : Command line warning D9002 : ignoring unknown option '-G5'
cl : Command line warning D9002 : ignoring unknown option '-QI0f'
cl : Command line warning D9024 : unrecognized source file type 'Files\Microsoft
', object file assumed
cl : Command line warning D9027 : source file 'Files\Microsoft' ignored
cl : Command line warning D9024 : unrecognized source file type 'Visual', object
 file assumed
cl : Command line warning D9027 : source file 'Visual' ignored
cl : Command line warning D9024 : unrecognized source file type 'Studio', object
 file assumed
cl : Command line warning D9027 : source file 'Studio' ignored
cl : Command line warning D9024 : unrecognized source file type '9.0\VC\include'
, object file assumed
cl : Command line warning D9027 : source file '9.0\VC\include' ignored
cl : Command line warning D9024 : unrecognized source file type 'Files\Microsoft
', object file assumed
cl : Command line warning D9027 : source file 'Files\Microsoft' ignored
cl : Command line warning D9024 : unrecognized source file type 'Visual', object
 file assumed
cl : Command line warning D9027 : source file 'Visual' ignored
cl : Command line warning D9024 : unrecognized source file type 'Studio', object
 file assumed
cl : Command line warning D9027 : source file 'Studio' ignored
cl : Command line warning D9024 : unrecognized source file type '9.0\VC\mfc\incl
ude', object file assumed
cl : Command line warning D9027 : source file '9.0\VC\mfc\include' ignored
kin_hexapodFullTest.c
Making .\lib\libhexapodFullTest.lib ...
   Creating library .\lib\libhexapodFullTest.lib and object .\lib\libhexapodFul
Test.exp
.\lib\libhexapodFullTest.dll is up-to-date
```

Figure F.5. VS command prompt after executing nmake all command

- After seeing the ***up-to-date*** message, 2 different files are created in the ***lib*** folder created in the beginning. One of these files end with ***.lib*** extension whereas the other has ***.lib.manifest*** in the end.
- If the completed message as above cannot be observed, the prompt window will show the errors in C-files. These errors can be semantic or syntax related; and thus, they should be corrected accordingly.

COMPILATION IN 32-BIT OPERATING SYSTEMS

- Go to "***C:\Program Files\Dassault Systemes\B21\intel_a\startup\DNBUserKinematics***". The directory may vary depending on the user.
- At this point make sure that the ***DNBUserKinematics*** folder has ***glaux.lib*** file. If not, copy and paste the file directly without making any changes on the ***glaux.lib*** file. This file ***glaux.lib*** is a standard library file and it can be directly found on the web.
- Create a folder named ***lib*** in the same directory. Copy and paste the C-file created in this directory as well. The result should look like this



Address C:\Program Files\Dassault Systemes\B21\intel_a\startup\DNBUserKinematics

Name	Size	Type
exmpl_data		File Folder
include		File Folder
lib		File Folder
glaux.lib	1 171 KB	Object File Library
kin_hexapodFullTest.c	18 KB	C Source
makefile	3 KB	File

Figure F.6. DNBUserKinematics folder

- At this point, necessary changes in ***makefile*** will be made. Open ***makefile*** in ***VS***, ***WordPad*** or ***NotePad.***
- In ***makefile***, go to lines 1-5 (which specify the names of library and locations that will be created after compilation) and find the following code and change as following

```
#
# library name and locations
#
DEST                = .\lib
LIBRARY             = libhexapodFullTest
```

- Go to line 10 and paste directory of the VisualStudio. It should look like as

```
#
# set up the MS Visual C++ compiler
#
MS_LOC= c:\Program Files (x86)\Microsoft Visual Studio 9.0\VC
```

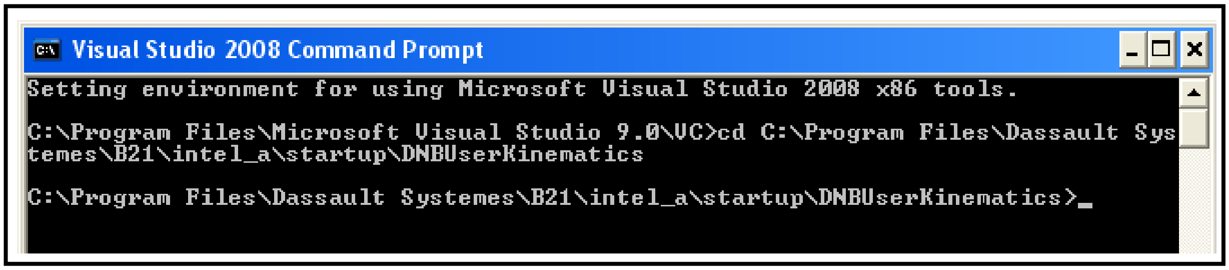
- **OBS!** For other versions of VS, change directory accordingly.
- On line 58, the name of the C-file created before should be specified. Thus it should look like as following(notice that it should end with *obj* extension)

```
#
# files to compile
#
OBJS                = \
          kin_hexapodFullTest.obj
```

- When the changes are made, save and close the ***makefile***.

- Now, proceed to compilation operation. First, open the VS command prompt and enter the following command

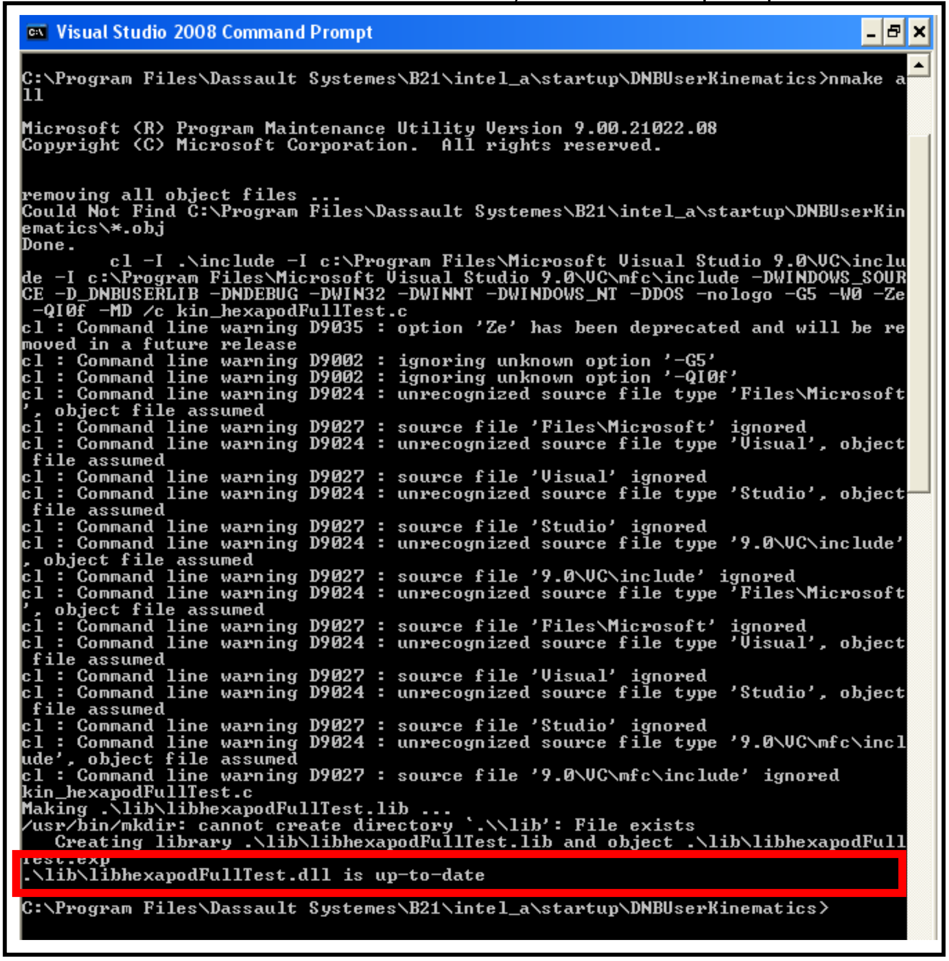
  cd C:\Program Files\Dassault Systemes\B21\win_b64\startup\DNBUserKinematics



```
Setting environment for using Microsoft Visual Studio 2008 x86 tools.

C:\Program Files\Microsoft Visual Studio 9.0\VC>cd C:\Program Files\Dassault Sys
temes\B21\intel_a\startup\DNBUserKinematics

C:\Program Files\Dassault Systemes\B21\intel_a\startup\DNBUserKinematics>_
```

Figure F.7. VS command prompt with folder destination

- Now, the compilation operation can be done. Again remain under the same directory in command prompt, and write ***nmake all.*** If C-file has no errors, the command prompt will look like



```
C:\Program Files\Dassault Systemes\B21\intel_a\startup\DNBUserKinematics>nmake a
ll

Microsoft (R) Program Maintenance Utility Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.


removing all object files ...
Could Not Find C:\Program Files\Dassault Systemes\B21\intel_a\startup\DNBUserKin
ematics\*.obj
Done.
        cl -I .\include -I c:\Program Files\Microsoft Visual Studio 9.0\VC\inclu
de -I c:\Program Files\Microsoft Visual Studio 9.0\VC\mfc\include -DWINDOWS_SOUR
CE -D_DNBUSERLIB -DNDEBUG -DWIN32 -DWINNT -DWINDOWS_NT -DDOS -nologo -G5 -W0 -Ze
 -QI0f -MD /c kin_hexapodFullTest.c
cl : Command line warning D9035 : option 'Ze' has been deprecated and will be re
moved in a future release
cl : Command line warning D9002 : ignoring unknown option '-G5'
cl : Command line warning D9002 : ignoring unknown option '-QI0f'
cl : Command line warning D9024 : unrecognized source file type 'Files\Microsoft
', object file assumed
cl : Command line warning D9027 : source file 'Files\Microsoft' ignored
cl : Command line warning D9024 : unrecognized source file type 'Visual', object
 file assumed
cl : Command line warning D9027 : source file 'Visual' ignored
cl : Command line warning D9024 : unrecognized source file type 'Studio', object
 file assumed
cl : Command line warning D9027 : source file 'Studio' ignored
cl : Command line warning D9024 : unrecognized source file type '9.0\VC\include'
, object file assumed
cl : Command line warning D9027 : source file '9.0\VC\include' ignored
cl : Command line warning D9024 : unrecognized source file type 'Files\Microsoft
', object file assumed
cl : Command line warning D9027 : source file 'Files\Microsoft' ignored
cl : Command line warning D9024 : unrecognized source file type 'Visual', object
 file assumed
cl : Command line warning D9027 : source file 'Visual' ignored
cl : Command line warning D9024 : unrecognized source file type 'Studio', object
 file assumed
cl : Command line warning D9027 : source file 'Studio' ignored
cl : Command line warning D9024 : unrecognized source file type '9.0\VC\mfc\incl
ude', object file assumed
cl : Command line warning D9027 : source file '9.0\VC\mfc\include' ignored
kin_hexapodFullTest.c
Making .\lib\libhexapodFullTest.lib ...
/usr/bin/mkdir: cannot create directory `.\\lib': File exists
   Creating library .\lib\libhexapodFullTest.lib and object .\lib\libhexapodFull
Test.exp
.\lib\libhexapodFullTest.dll is up-to-date

C:\Program Files\Dassault Systemes\B21\intel_a\startup\DNBUserKinematics>
```

Figure F.8. VS command prompt after executing nmake all command

- After seeing the ***up-to-date*** message, 2 different files are created in the ***lib*** folder created in the beginning. One of these files end with ***.lib*** extension whereas the other has ***.lib.manifest*** in the end.
- If the completed message as above cannot be observed, the prompt window will show the errors in C-files. These errors can be semantic or syntax related.

APPENDIX G: MATLAB FUNCTIONS

HEXAPOD INVERSE KINEMATICS

```
function [ ] =hexapod( px,py,pz,tex,tey,tez,T6)
%Enter your tex,tey,tez values that are rotations of the mobile platform in
%degrees. px,py and pz values are the translations in mm. T6 and TCP
%matrices should be created in the workspace and entered with their names
%into the function.
%--------------------------------------------------------------------------
%Creation of necessary symbols and conversions from degrees to radians.
syms x y z tx ty tz;
tex=tex*pi/180;
tey=tey*pi/180;
tez=tez*pi/180;
%--------------------------------------------------------------------------
%Fix values are assigned to the vectors where L.b is the base vector in
%base coordinate frame whereas L.tToTCPs are the vectors that connect the
mobiles
%cooridnate frame to upper attachment points.

L1b=[31;118.156;40.205;1];
L1tToTCP=[31;48.799;-31.45;1];
L2b=[-31;118.156;40.205;1];
L2tToTCP=[-31;48.799;-31.45;1];
L3b=[-117.826;-32.231;40.205;1];
L3tToTCP=[-57.761;2.447;-31.45;1];
L4b=[-86.826;-85.925;40.205;1];
L4tToTCP=[-26.761;-51.246;-31.45;1];
L5b=[86.826;-85.925;40.205;1];
L5tToTCP=[26.761;-51.246;-31.45;1];
L6b=[117.826;-32.231;40.205;1];
L6tToTCP=[57.761;2.447;-31.45;1];
Lref=399.413;

%--------------------------------------------------------------------------
%Creation of transformation matrices for the given values of TCP for T6 and
%TCP matrices.
%R symbolizes rotation here.

R_T6=subs(T6,x,px);
R_T6=subs(R_T6,y,py);
R_T6=subs(R_T6,z,pz);
R_T6=subs(R_T6,tx,tex);
R_T6=subs(R_T6,ty,tey);
R_T6=subs(R_T6,tz,tez);
%--------------------------------------------------------------------------
% Finding leg lengths by using T6 matrix
L1_t6=R_T6*L1tToTCP-L1b;
L2_t6=R_T6*L2tToTCP-L2b;
L3_t6=R_T6*L3tToTCP-L3b;
L4_t6=R_T6*L4tToTCP-L4b;
L5_t6=R_T6*L5tToTCP-L5b;
L6_t6=R_T6*L6tToTCP-L6b;
%--------------------------------------------------------------------------
```

```
%Vectors to describe upper attachment points from mobile platform with
%respect to the base coordinate system

L1_tTCP=R_T6(1:3,1:3)*L1tToTCP(1:3);
L2_tTCP=R_T6(1:3,1:3)*L2tToTCP(1:3);
L3_tTCP=R_T6(1:3,1:3)*L3tToTCP(1:3);
L4_tTCP=R_T6(1:3,1:3)*L4tToTCP(1:3);
L5_tTCP=R_T6(1:3,1:3)*L5tToTCP(1:3);
L6_tTCP=R_T6(1:3,1:3)*L6tToTCP(1:3);

%---------------------------------------------------------------------------
%total lenght - standard length = joint values for T6 matrix
J1_t6=norm(L1_t6)-Lref;
J2_t6=norm(L2_t6)-Lref;
J3_t6=norm(L3_t6)-Lref;
J4_t6=norm(L4_t6)-Lref;
J5_t6=norm(L5_t6)-Lref;
J6_t6=norm(L6_t6)-Lref;
%---------------------------------------------------------------------------
%Displaying the results
Leg_lengths=[J1_t6;J2_t6;J3_t6;J4_t6;J5_t6;J6_t6]
Coordinates_T6=[R_T6*L1tToTCP,R_T6*L2tToTCP,R_T6*L3tToTCP,R_T6*L4tToTCP,R_T6*
L5tToTCP,R_T6*L6tToTCP]
R_T6

end
```

FLEXAPOD INVERSE KINEMATICS

```
function [ ] =flexapod( px,py,pz,tex,tey,tez,T6)
%Enter your tex,tey,tez values that are rotations of the mobile platform in
%degrees. px,py and pz values are the translations in mm. T6 and TCP
%matrices should be created in the workspace and entered with their names
%into the function.
%--------------------------------------------------------------------------
%Creation of necessary symbols and conversions from degrees to radians.
syms x y z tx ty tz;
tex=tex*pi/180;
tey=tey*pi/180;
tez=tez*pi/180;
%--------------------------------------------------------------------------
%Fix values are assigned to the vectors where L.b is the base vector in
%base coordinate frame whereas L.tToTCPs are the vectors that connect the
mobiles
%cooridnate frame to upper attachment points.

L1b=[-132.5;26;58.5;1];
L1tToTCP=[-48.767;32.466;-75;1];
L2b=[43.733;127.748;58.5;1];
L2tToTCP=[-3.733;58.466;-75;1];
L3b=[88.767;101.748;58.5;1];
L3tToTCP=[52.5;26;-75;1];
L4b=[88.767;-101.748;58.5;1];
L4tToTCP=[52.5;-26;-75;1];
L5b=[43.733;-127.748;58.5;1];
L5tToTCP=[-3.733;-58.466;-75;1];
L6b=[-132.5;-26;58.5;1];
L6tToTCP=[-48.767;-32.466;-75;1];
Lref=376.5;

%--------------------------------------------------------------------------
%Creation of transformation matrices for the given values of TCP for T6 and
%TCP matrices.
%R symbolizes rotation here.

R_T6=subs(T6,x,px);
R_T6=subs(R_T6,y,py);
R_T6=subs(R_T6,z,pz);
R_T6=subs(R_T6,tx,tex);
R_T6=subs(R_T6,ty,tey);
R_T6=subs(R_T6,tz,tez);
%--------------------------------------------------------------------------
% Finding leg lengths by using T6 matrix
L1_t6=R_T6*L1tToTCP-L1b;
L2_t6=R_T6*L2tToTCP-L2b;
L3_t6=R_T6*L3tToTCP-L3b;
L4_t6=R_T6*L4tToTCP-L4b;
L5_t6=R_T6*L5tToTCP-L5b;
L6_t6=R_T6*L6tToTCP-L6b;
%--------------------------------------------------------------------------
%Vectors to describe upper attachment points from mobile platform with
%respect to the base coordinate system
```

```
L1_tTCP=R_T6(1:3,1:3)*L1tToTCP(1:3);
L2_tTCP=R_T6(1:3,1:3)*L2tToTCP(1:3);
L3_tTCP=R_T6(1:3,1:3)*L3tToTCP(1:3);
L4_tTCP=R_T6(1:3,1:3)*L4tToTCP(1:3);
L5_tTCP=R_T6(1:3,1:3)*L5tToTCP(1:3);
L6_tTCP=R_T6(1:3,1:3)*L6tToTCP(1:3);

%---------------------------------------------------------------------------
%total lenght - standard length = joint values for T6 matrix
J1_t6=norm(L1_t6)-Lref;
J2_t6=norm(L2_t6)-Lref;
J3_t6=norm(L3_t6)-Lref;
J4_t6=norm(L4_t6)-Lref;
J5_t6=norm(L5_t6)-Lref;
J6_t6=norm(L6_t6)-Lref;
%---------------------------------------------------------------------------
%Displaying the results
Leg_lengths=[J1_t6;J2_t6;J3_t6;J4_t6;J5_t6;J6_t6]
Coordinates_T6=[R_T6*L1tToTCP,R_T6*L2tToTCP,R_T6*L3tToTCP,R_T6*L4tToTCP,R_T6*
L5tToTCP,R_T6*L6tToTCP]
R_T6

end
```

EXECHON INVERSE KINEMATICS

```
function [ ] =exechon( px,py,pz,tex,tey,tez,T6)
%Enter your tex,tey,tez values that are rotations of the mobile platform in
%degrees. px,py and pz values are the translations in mm. T6 and TCP
%matrices should be created in the workspace and entered with their names
%into the function.
%--------------------------------------------------------------------------
%Creation of necessary symbols and conversions from degrees to radians.
format short e
syms x y z tx ty tz;
tex=tex*pi/180;
tey=tey*pi/180;
tez=tez*pi/180;
%--------------------------------------------------------------------------
%Fix values are assigned to the vectors where L.b is the base vector in
%base coordinate frame whereas L.tToTCPs are the vectors that connect the
mobiles
%cooridnate frame to upper attachment points.

L1b=[420;0;0;1];
L1tToTCP=[173;-50;485;1];
L2b=[-420;0;0;1];
L2tToTCP=[-173;-50;485;1];
L3b=[0;670;0;1];
L3tToTCP=[-0;173;485;1];
Lref12=803.887;
Lref3=886.021;

%--------------------------------------------------------------------------
%Creation of transformation matrices for the given values of TCP for T6 and
%TCP matrices.
%R symbolizes rotation here.

R_T6=subs(T6,x,px);
R_T6=subs(R_T6,y,py);
R_T6=subs(R_T6,z,pz);
R_T6=subs(R_T6,tx,tex);
R_T6=subs(R_T6,ty,tey);
R_T6=subs(R_T6,tz,tez);
%--------------------------------------------------------------------------
% Finding leg lengths by using T6 matrix
L1_t6=R_T6*L1tToTCP-L1b;
L2_t6=R_T6*L2tToTCP-L2b;
L3_t6=R_T6*L3tToTCP-L3b;
%--------------------------------------------------------------------------
%Vectors to describe upper attachment points from mobile platform with
%respect to the base coordinate system

L1_tTCP=R_T6(1:3,1:3)*L1tToTCP(1:3);
L2_tTCP=R_T6(1:3,1:3)*L2tToTCP(1:3);
L3_tTCP=R_T6(1:3,1:3)*L3tToTCP(1:3);
```

```
%--------------------------------------------------------------------------
%total lenght - standard length = joint values for T6 matrix
J1_t6=norm(L1_t6)-Lref12;
J2_t6=norm(L2_t6)-Lref12;
J3_t6=norm(L3_t6)-Lref3;
J1_t6_1=-norm(L1_t6)-Lref12;
J2_t6_1=-norm(L2_t6)-Lref12;
J3_t6_1=-norm(L3_t6)-Lref3;

%--------------------------------------------------------------------------
%Displaying the results
Leg_lengths=[J1_t6 J1_t6_1;J2_t6 J2_t6_1;J3_t6 J3_t6_1]
Coordinates_T6=[R_T6*L1tToTCP,R_T6*L2tToTCP,R_T6*L3tToTCP]
R_T6

end
```

GANTRY-TAU FORWARD KINEMATICS

```
function []=tau_forward(p1,p2,p3)
%The coordinates of presented vectors with joint values of p1,p2 and p3
inserted.
P1=[0;0;-p1];
P2=[-1100;700;-p2];
P3=[-2200;0;-p3];
d1=[-96.569;-185;-400];
d2=[0;-322.843;-173.726];
d3=[96.569;0;-400];
n1=[224.99;-240.001;171.568];
n2=[0;39.705;336.862];
n3=[-182.574;-80;213.994];
L1=1500;
L2=1499.775;
syms x y z
T=[x;y;z];

%The equations A1,B1 and C1
A=T+n1-(P1+d1);
B=T+n2-(P2+d2);
C=T+n3-(P3+d3);
A1=A(1,1)^2+A(2,1)^2+A(3,1)^2-L1^2;
B1=B(1,1)^2+B(2,1)^2+B(3,1)^2-L1^2;
C1=C(1,1)^2+C(2,1)^2+C(3,1)^2-L2^2;

%The solution
[x,y,z]=solve(A1,B1,C1);
T=[x(1) y(1) z(1);x(2) y(2) z(2)];
double(T)
```

GANTRY-TAU INVERSE KINEMATICS

```
function [ ] =tau( px,py,pz,tex,tey,tez,T6)
syms x y z tx ty tz;
%The radian to degree conversion
tex=tex*pi/180;
tey=tey*pi/180;
tez=tez*pi/180;
%The coordinates of presented vectors.
P1=[0;0;0;1];
P2=[-1100;700;0;1];
P3=[-2200;0;0;1];
d1=[-96.569;-185;-400;1];
d2=[0;-322.843;-173.726;1];
d3=[96.569;0;-400;1];
n1=[224.99;-240.001;171.568;1];
n2=[0;39.705;336.862;1];
n3=[-182.574;-80;213.994;1];
%Reference leg lengths
Lref=1500;
Lref2=1499.775;
%Inserting TCP values to transformation matrix
R_T6=subs(T6,x,px);
R_T6=subs(R_T6,y,py);
R_T6=subs(R_T6,z,pz);
R_T6=subs(R_T6,tx,tex);
R_T6=subs(R_T6,ty,tey);
R_T6=subs(R_T6,tz,tez);
%Multiplication of mobile platform vectors with transformation matrix
NT1=R_T6*n1;
NT2=R_T6*n2;
NT3=R_T6*n3;
%Applying equations for each joint
J11=-1*(NT1(3,1)-d1(3,1)+sqrt(Lref^2-(P1(1,1)+d1(1,1)-NT1(1,1))^2-
(P1(2,1)+d1(2,1)-NT1(2,1))^2));
J12=-1*(NT1(3,1)-d1(3,1)-sqrt(Lref^2-(P1(1,1)+d1(1,1)-NT1(1,1))^2-
(P1(2,1)+d1(2,1)-NT1(2,1))^2));
J21=-1*(NT2(3,1)-d2(3,1)+sqrt(Lref^2-(P2(1,1)+d2(1,1)-NT2(1,1))^2-
(P2(2,1)+d2(2,1)-NT2(2,1))^2));
J22=-1*(NT2(3,1)-d2(3,1)-sqrt(Lref^2-(P2(1,1)+d2(1,1)-NT2(1,1))^2-
(P2(2,1)+d2(2,1)-NT2(2,1))^2));
J31=-1*(NT3(3,1)-d3(3,1)+sqrt(Lref2^2-(P3(1,1)+d3(1,1)-NT3(1,1))^2-
(P3(2,1)+d3(2,1)-NT3(2,1))^2));
J32=-1*(NT3(3,1)-d3(3,1)-sqrt(Lref2^2-(P3(1,1)+d3(1,1)-NT3(1,1))^2-
(P3(2,1)+d3(2,1)-NT3(2,1))^2));

%Displaying results
R_T6
%Joint values
Joint_values=[J11,J21,J31;J11,J21,J32;J11,J22,J31;J11,J22,J32;J12,J21,J31;J12
,J21,J32;J12,J22,J31;J12,J22,J32];
Joint_values
end
```