COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

HUMANOID ROBOT LILLI

Master Thesis

2020 Bc. Gabriel Halasi

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

HUMANOID ROBOT LILLI

Master Thesis

Study programme:	Aplied informatics
Study field:	2511 Aplied informatics
Department:	Department of Applied Informatics
Supervisor:	Mgr. Pavel Petrovič, PhD.

Bratislava, 2020

Bc. Gabriel Halasi

41590314

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Gabriel Halasi
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Humanoid Robot Lilli

Annotation: Robot Lilli has been presented at Maker Faire 2018 in Vienna by its author Per R. Ø. Salkowitsch. It is a humanoid robot with 25 degrees of freedom constructed of laser-cut plywood pieces. There is no control sotware for the robot available at the moment. The aim of the diploma thesis is to study and implement algorithms that will allow the robot to move in its environment including inverse kinematics and utilizing the Machine Learning algorithms. It is expected that the student will develop a simulated model of the robot and verifies the algoithms both in simulation and on the real robot.

Literature: R.Siegwart et.al: Introduction to Autonomous Mobile Robots, The MIT Press, 2011.
H. Choset et.al: Principles of Robot Motion, Theory, Algorithms, and Implementations, The MIT Press, 2005.

Keywords: humanoid robot, inverse kinematics, machine learning, simulation

Supervisor:	Mgr. Pavel Petrovič, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	26.09.2018
Approved:	31.10.2018

prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

Student

Supervisor

41590314

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Gabriel Halasi
Študijný program:	aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	aplikovaná informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Humanoid Robot Lilli
Humanoidný Robot Lilli

Anotácia: Robot Lilli predstavil na viedenskom podujatí Maker Faire 2018 jeho autor Per R. Ø. Salkowitsch. Ide o humanoidného robota s 25 stupnami voľnosti vytvoreného z dielov vyrezaných z preglejky laserom. K robotu zatiaľ neexistuje obslužný softvér. Cieľom diplmovej práce bude preskúmať a implementovať algoritmy, pomocou ktorých sa robot bude vedieť pohybovať vo svojom prostredí, vrátane inverznej kinematiky a využitia algoritmov strojového učenia. Predpokladá sa vytvorenie modelu robota pre simuláciu a otestovanie algoritmov v simulácii i na reálnom robotovi.

Literatúra: R.Siegwart et.al: Introduction to Autonomous Mobile Robots, The MIT Press, 2011.
H. Choset et.al: Principles of Robot Motion, Theory, Algorithms, and Implementations, The MIT Press, 2005.

Kľúčové slová: humanoidný robot, inverzná kinematika, strojové učenie, simulácia

Vedúci:	Mgr. Pavel Petrovič, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	prof. Ing. Igor Farkaš, Dr.
Dátum zadania:	26.09.2018

Dátum schválenia: 31.10.2018

prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

....
študent

....
vedúci práce

Acknowledgement

In the first place, I want to thank my supervisor, Mgr. Pavel Petrovič, PhD., for the support, willingness and encouraging words that were helpful for me. Next, I also want to thank my girlfriend, family, relatives and friends for helping and enduring many of these difficult moments.

Abstract

The Diploma thesis is about a Humanoid robot Lilli and provides an insight into the history of humanoid robots, the robot shapes, types of robots and their degree of freedom. The work deals with the construction of a humanoid robot from the basics, which includes various CAD systems to model any type of robot, the joint selection and the model definition in unified robot description format. The thesis offers a comparison of simulators and then their use for motion simulations. Furthermore, in this work we managed to build a clean model, calculate the center of mass and the rotational inertia for all the separate links and to set the position of the joints. After that, we successfully created the URDF structure, which fully describes the robot definition except the closed loop chain. The second approach of this thesis are simulations in the CoppeliaSim environment. It explains the scene, model, simulation control using remote API and embedded scripts. In the research part of the work we describe the forward and inverse kinematics problems. The last part shows the control of a kinematic chain, where we successfully reached an arm motion with Pseudo inverse and DLS calculation methods, which are part of the inverse kinematics solver.

Keywords: humanoid robot, CAD systems, clean model, 3D model, URDF, CoppeliaSim, remote API, inverse kinematics

Abstrakt

Diplomová práca s názvom Humanoid robot Lilli poskytuje stručný prehľad o histórii humanoidných robotov, o tvaroch a typoch robotov, o ich stupňoch voľnosti pohybu. Práca sa zaoberá vybudovaním humanoidného robota od základov, čo zahŕňa rôzne CAD systémy, pomocou ktorých sa dá vymodelovať hocijaký robot, výber typov servomotorov a definícia modelu v unifikovanom opisnom formáte robota. Práca ponúka porovnanie simulátorov a ich použitie pri simulácii pohybu. Následne sa nám v práci podarilo vybudovať čistý model, vypočítať ťažisko a krútiaci moment pre všetky samostatné linky a nastaviť pozíciu servomotorov. Ďalej sme úspešne vytvorili URDF štruktúru, ktorá plne opisuje definíciu robota okrem uzatvorených reťazcov. Druhá časť práce sa zaoberá simuláciou v prostredí CoppeliaSim. Vysvetľuje scénu, model, riadenie simulácie pomocou remote API a vstavaných scriptov. V časti výskum popisujeme problémy doprednej a inverznej kinematiky. V poslednej časti je popísané riadenie kinematického reťazca, pri ktorom sme úspešne dosiahli pohyb ramenom pomocou kalkulačných metód Pseudo inverse a DLS.

popis robota v zjednotenej podobe alebo porovnávanie simulátorov a následne ich použitie na simulácie pohybu. Ďalej táto práca popisuje tiež hmotnostné vlastnosti robota ako je vyrátanie ťažiska, moment zotrvačnosti a rieši problematiky doprednej a inverznej kinematiky. Následne práca poukáže na to, ako vybudovať 3D model správne pre vybraný simulátor a uložiť do štruktúry URDF, ktorá je zjednotená. Druhá časť práce sa zaoberá simuláciou v prostredí CoppeliaSim. Vysvetľuje scénu, model, riadenie simulácie pomocou remote API a vstavaných scriptov. V poslednej časti je popísané riadenie kinematického reťazca pomocou inverznej kinematiky.

Kľúčové slová: humanoid robot, CAD programy, čistý model, 3D model, URDF, CoppeliaSim, remote API, inverzná kinematika

Contents

List of Figures

Chapter 1

Introduction

Humanoid robotics is a developing research field and it is becoming more and more popular which can be seen in the number of recent researches.

This branch is dealing with building humanoid robots in order to understand the processing of human-like information. This includes the form of appearance, solving the human motion, interaction and other features. The research in humanoid robotics disposes many uncovered issues and describes solutions to classical problems of control theory or in artificial intelligence. Nowadays there exist many researches such as Haikawa & Takenaka 1998 [Beh08] or Inaba & Inoue 1998 [Beh08] which describe integrated humanoid robots. In addition, except these researches there exist conferences dedicated only to humanoid systems such as the International Symposium on Humanoid Robots (HURO) [Beh08], which was first held at Waseda University [Beh08]. Major importance for advances of the field is with no doubt the availability of reproducible humanoid robots systems, which have been used in the last years as common hardware and software platforms to support humanoids research. This news are reflected by the firmly established annual IEEE-RAS International Conference [Beh08] on Humanoid Robots, which is an internationally recognized prime event of the humanoid robotics community.

Humanoid robots are being used in the inspection, maintenance and disaster response at power plants to relieve human workers of laborious and dangerous tasks. These robots are a relatively new form of professional service robots. While long-dreamt about, they are now starting to become commercially viable in a wide range of applications. The humanoid robots market is poised for significant growth. It is projected that the market for humanoid robots will be valued at 3.9 Billion dollars in 2023.

In this research we will build our humanoid robot from the start. That means, we will create a 3D model of the robot and after that we would like to structure that model into a unified robot description format. In order to build the 3D model we need to define the robot links as a mesh or a primitive shape, which are adapted to our idea. Next we will substitute the robot servomotors with the correct type of joint and we must find a CAD system to define the relationship between the links and the joints. Except of that, we will calculate the links center of mass, rotational inertia and mass, which are important to the physics engine. Depending of the robot model we will select a unified robot structure to describe it. The second approach of the research will be to demonstrate the forward and inverse kinematics to solve the joint variables. In this step we will use the inverse kinematics solver of the simulator to simulate a kinematic chain motion.

Chapter 2

Motivation

This idea started in 2018 at the Maker Faire event, when Per R. Ø. Salkowitsch showed robot Lilli to the audience. We talk about a humanoid robot, which has twenty-five degree of freedom and its parts were carved out from plywood sheet with laser. At that time, the servicing system did not exist in the robot. Our goal will be to explore and implement algorithms, which can serve to control the robot's motion in his own environment. This research must include inverse kinematics and must utilize algorithms of machine learning. It is assumed that we need to create a 3D robot model for simulation purposes and testing algorithms by simulations and after that on the real robot.

Chapter 3

Research

In this section we will discuss the existing solutions, systems, solvers and researches which help us to understand how humanoid robots work, how to start building these robot systems or how to solve its physics or inverse kinematics problems. First, we need to introduce the humanoid robot, its past and we will talk a little bit about humanoid types. After that, we will see how to construct a 3D model of the real robot and this construction we will use to create simulation in robot simulator. At the end, you will see the research about robot simulators, its advantages and disadvantages and which robot simulator is selected for the simulation. Next part is dedicated to physics and inverse kinematics of humanoid robots. This includes existing researches, solutions and problems of kinematics.

3.1 Humanoid robots

A humanoid robot is generally defined as a programmable machine, which was built to mimic human motions and interactions. Humanoid robots need to solve two types of challenges. First, they need to be able to learn, which includes many different kinds of cognitive capabilities - from recognition of new types of objects, and properties of their environments, learning new skills, new knowledge, improving communication abilities with humans, and second, they need to perform and carry out physical work, for example moving objects or take care of its own locomotion and movements One of them is learning new information and the second is carrying out physical work, for example moving objects. Let us see the historical development of humanoid robots.

3.1.1 History and overview

The history of mechanical systems is wide. These mechanics imitate the human behaviour. Even several hundreds years ago, people were building mechanical devices that were imitating human behavior. For instance, one of the earliest forms of humanoids was created in 1495 by Leonardo Da Vinci [Beh08]. The robot could stand, sit, raise its visor and individually move its arms. The entire robot was operated by a number of pulleys and cables.

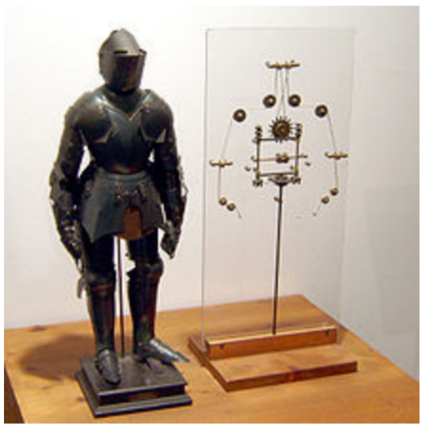


Figure 3.1: Da Vinci´s humanoid automata adopted from [Bor16]

The word robot was first presented in the literary work RUR by the

Czech author Karel Capek in 1921. The mechanical servant in the play had a humanoid appearance. After that in 1926 Maria was the first humanoid robot which appeared in the movie.

In the second part of the 20th century the development in the information technology made it possible to include some important computations for the sensing, controlling and manipulating of the robots. Researchers developed many isolated systems for sensing and motion which included human abilities. In 1973 a robot called Wabot-1 [Bor16] was invented, which included previous systems. This robot was full-scale anthropomorphic robot able to walk on two legs.

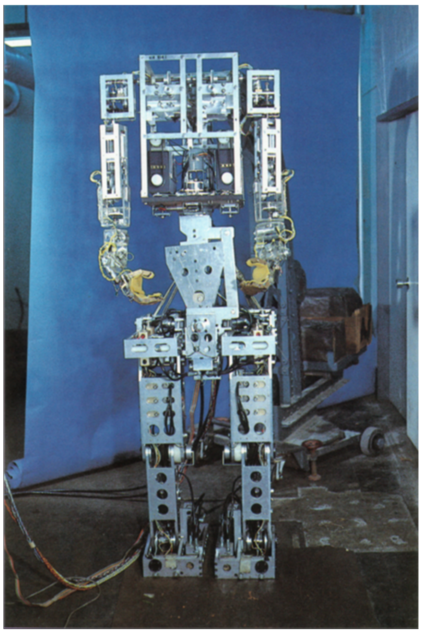


Figure 3.2: Wabot-1 full-scale anthropomorphic robot adopted from [Bor16]

After these inventions in 1996 Honda revealed its Humanoid P2 [Bor16] robot, which was able to walk steadily on two feet. It was the first self-contained full-body humanoid. In the U.S., researchers completed a full-scale android body. Another invention was constructed by company Sony. They created a small humanoid robot which was able to recognize faces, could express emotions and could walk on flat surface.This robot was called Qrio.

The following robots are very innovative and popular world wide:

- Sophia: She was created by Hanson robotics and can accomplish a wide range of human actions. It was the first robot citizen. The robot is able to make fifty facial expressions.
- The Kodomoroid TV Presenter: This humanoid robot's name is composed of two words: child-Kodomo and Android. This robot was invented by Japanese researchers and she speaks more languages, she is able to read news and give weather forecast.
- Jia Jia: They have been working on the model for three years before they introduced it at the University of Science and Technology of China. It has got limited motion and it can make conversation.

3.1.2 Robot shapes

At present, different humanoid robots exist world wide. Different means that they have different sizes and shapes. Furthermore, these robots have different behaviour. Some models are focusing only to head and face and other models have head with hands which are located on the static torso. At the end, there exist full-body humanoids on the wheels and without two feet. These differences are in play in the usage of robots mainly in mobility, actuating, motion of body or interaction with humans.

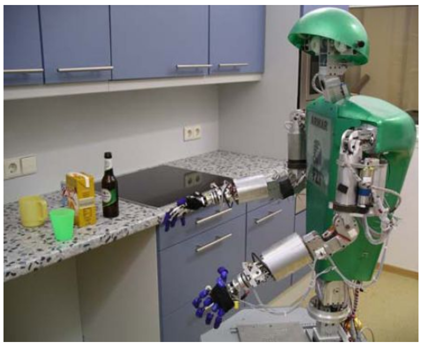


Figure 3.3: ARMAR-III upper body humanoid robot adopted from [Edw]

Upon humanoids we will usually classify them into two groups, into Androids or Gynoids. Design of an Android [Bor16] was made like a male human and gynoid [Bor16] emulates a female human.

Humanoids possess certain features. They have sensors that help them in sensing their environment. Some have cameras that allow them to see clearly. Motors placed at strategic points are what guides them in moving and making gestures. These motors are usually referred to as actuators.

3.1.3 Terms characteristic

Actuators - also known as motors, which help in motion and making expressions with moving humanoid body. Our body is dynamic and therefore we easily pick up things and manipulate them. Humanoid robots need to do the same things by simulating wide range of our actions. These actuators imitate the actions flexibly and effectively.

Sensors - humanoid robots use them to sense the environment around them. Robots need many sensors to carry out expressions correctly and without any damage. For example they need sensor for balanced movement or a sensor to hear instructions or facial sensors to make facial expressions.

AI-based interactions - human interactions with humanoids can be mimic limited. It can help artificial intelligence to understand human ramblings and give it back in the form of replies.

3.2 Humanoid robot model and simulators

In this part we will discuss the humanoid body movements, systems to draw the robot parts, robot description formats and simulators. What kind of solutions exist to create a 3D model and we will use this model for some simulations of movement.

3.2.1 Degree of Freedom

A humanoid robot was made to mimic humans, therefore its body parts like legs are able to move on its own. The number of Degree of Freedom (DOF) is given by the number of their joint actuators. According humans, humanoids´ body moves in next three planes [SRR11].

- transverse(axial) plane: This plane is an imaginary plane that divides the body into inferior and superior.
- frontal(coronal) plane: This plane is perpendicular to the sagittal plane and divides the body into front and back.
- sagittal plane: This plane divides the body into right and left sides.

Let us take an example from NAO [SRR11] robot model. In the picture we will see the location of actuators and their rotation movement. There are 3 DOF's located in each legs. These are in the hip (DOF 4), knee (DOF 3) and ankle (DOF 1) and moving leg in the sagittal plane. Actuators in the ankle (DOF 2) and hip (DOF 5) move on frontal plane and DOF 6 in the hip the leg moves in the transverse plane.

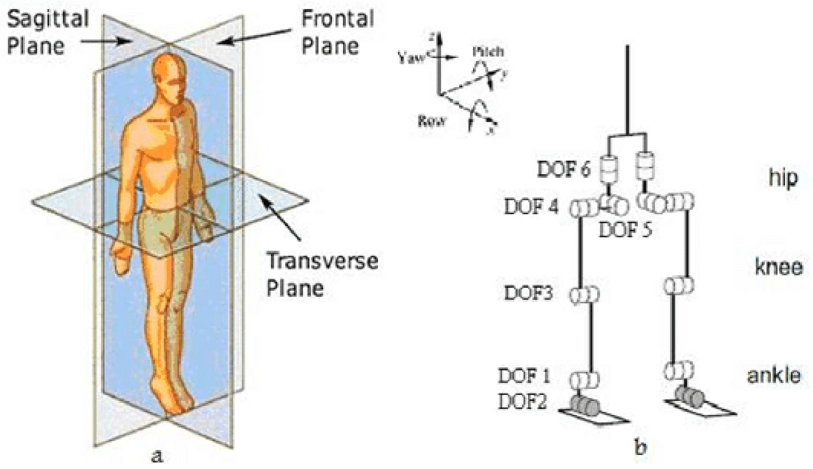


Figure 3.4: Human body planes and NAO joint's configuration adopted from [SRR11]

3.2.2 Modelling software

Further to make a 3D model of a humanoid robot we need a software where we can draw 2D and 3D shapes. In this part we will focus on three CAD software which we will use at implementation to draw robot body parts. Let us first evaluate the suitability of various software tools with respect to the relevance to our purpose

We will divide applications, which we use for 3D design, into 2 groups. First group is CAD (Computer aided design) software and the second is 3D modelling software. Even though both software make the same thing, CAD is usually used to create mechanical objects or industrial objects and 3D modelling grants wide artistic freedom.

Here we will discuss the CAD applications. The software contains a wide range of tools whose functions are used for industrial design, architecture or aerospace engineering. A CAD model includes information like dimensions, tolerance and material type. Many CAD applications can render and animate objects effectively to better visualise the product. You will choose to save this objects to STL (stereolithography) file format which is used by 3D printers.

We have chosen three CAD systems: TinkerCAD, SolidWorks and AutoCAD

TinkerCAD - This application is an online 3D app coming from Autodesk allowing you to create models from basic shapes and a lot of libraries exist which allow users to select the best shape and to manipulate with them. TinkerCAD has a direct cooperation with third party printing services and it is aimed for people with no experience.

SolidWorks - It was released by Dassault Systèmes. Software is a parametric featured-based model. In contrast with TinkerCAD this application has a design validation tool. SolidWorks uses a system called NURBS, which allows drawing detailed curvatures. A big disadvantage is the limited ability to import STL files. This sotware is aimed for professionals and that can be seen on the high price too.

AutoCAD - It is a software created by AutoDesk and it was the first CAD software released in 1982. AutoCAD is very popular across industries. This application is not only for 3D modelling but for 2D drafting, too. AutoCAD is aimed for people with mid level. Advantage of this software is that it can import and export STL format very well.

3.2.3 Robot description formats

People like making things easier everywhere. In modelling, people prefer human-readable and code-independent way of describing a robot body and physics or their cells. For example, in CAD systems it looks like this: one part of the body is 20cm right of another part and has square-mesh for display design. The robot description format allows the user to define the robot construction, physics, kinematics, shape etc. in structure, which is unified and easy to read. For the robot description purpose we have the following robot description formats:

- **URDF** (Unified robot description format): It is a standard robot description format which describes many robots very well. URDF [SZK17] is using XML specifications for describing the robot´s kinematic and dynamic properties, physical geometry, collision model and sensor locations. In this format a robot consists of links which are coupled with joints. Joints describe connection between them with many other information.

```
<?xml version="1.0">
<robot name="my_robot">
   <link name="link_name">
      <visual>
         <origin rpy="1.57025 0 0" xyz="0 0 0" />
      </visual>
      ...
   </link>
   <joint name="joint_name" type="fixed">
      ...
   </joint>
   <plugin filename="filename.ext" name="my_plugin" />
   ...
</robot>
```

- **SDF** [SZK17] (Simulation description format): It is based on XML format and it was originally created for Gazebo. Disadvantage of this format is that it is only usable in Gazebo environment. SDF describes objects and environment for robot simulators.

```
<?xml version="1.0">
<sdf version="1.5">
  <word name="default">
    <physics type="ode">
      ...
    </physics>
    <scene>
      ...
    </scene>
    <light>
      ...
    </light>
  </word>
  <model name="cube">
    <pose>0 0 0.5 0 0 0</pose>
    <static>false</static>
    <link name="link_name">
      <pose>0 0 0 0 0 0</pose>
      ...
    </link>
    <joint type="fixed" name="joint_name">
      ...
    </joint>
    <plugin filename="filename.ext" name="my_plugin" />
```

```
      ...
   </model>
</robot>
```

- **XACRO**: Xacro is a macro language based on XML structure. This format expands the large XML files into small parts by using macros. This format is used in large documents similarly to robot descriptions. Sometimes Xacro is used to simplify URDF format. We show a peace of Xacro code adapt from [SG].

```
<xacro:macro name="pr2_arm" params="suffix parent reflect">
<pr2_upperarm suffix="${suffix}" reflect="${reflect}"
   parent="${parent}" />
<pr2_forearm suffix="${suffix}" reflect="${reflect}"
   parent="elbow_flex_${suffix}" />
</xacro:macro>

<xacro:pr2_arm suffix="left" reflect="1" parent="torso" />
<xacro:pr2_arm suffix="right" reflect="-1" parent="torso" />
```

3.3 Robot simulators and comparison

Robot simulators are software which allow to simulate the real machine interactions in the selected environment. Usually a user creates the robot body with links, actuators and sensors and by a camera he follows the robot interactions during the simulation. In simulations the simulator contains the physics engine which would concur with the physics of the real world. Many robot simulators attempt a 3D view, which allows for user to follow up the robot behaviour in the selected environment. If the user sets the right settings for the robot then the real robot should have an identical behaviour. Let us see three robot simulators, how they work and what are their advantages or disadvantages.

3.3.1 V-REP simulator

V-REP [PGPW18] is a very feature-rich simulation environment that contains large and small scenes and a model editor. It has rich libraries to produce models and the biggest advantage is that V-REP allows a real-time mesh manipulation. Another advantage of V-REP is that it offers free licence for education purpose. This simulator is available for MacOS, Linux and Windows. In case of built-in capabilities V-REP includes many default engines such as Bullet 2.78, Bullet 2.83, ODE, Vortex and Newton. After starting the simulator we will see a code and a scene editor. The simulator has several aspects to control the simulation. One of these aspects are embedded scripts. It is very flexible, it has the best compatibility in terms of other installations and the embedded script is written in Lua programming language. Other popular aspects are ROS node, remote API or different plugins. From robot and model side, V-REP provides a wide collection of robots, actuators and sensors. The application allows us to manipulate and to simplify the model. The user has various options to create runnable scripts. Besides that V-REP includes plug-ins, ROS nodes and accepts Remote API connection.

3.3.2 Gazebo robot simulator

Gazebo [PGPW18] is used for bigger simulations like V-REP but the interface and the robot models are simpler. This simulator is available for MacOS, Linux and Windows too. As regards to built-in capabilities Gazebo allows only one physics engine end that is ODE. Next disadvantage in contrast with V-REP is that gazebo allows to import meshes as single objects, but these objects cannot be changed. The user can find lower number of libraries of default robot models but their documentations are defective. In case of programming methods Gazebo has a scene and a code editor too like V-REP, but it has fewer options for programming functionality. This application supports compiled C++ plug-ins or ROS programs. Gazebo has relatively good documentation and there exist many tutorials to learn how to make simulations but its documentation is a little behind V-REP simulator. An advantage is that creators will fully support the simulator in the feature and the development road map is available on their official website.

3.3.3 ARGoS robot simulator

ARGoS [PGPW18] was not really designed for big simulations, but it is a good choice for swarm robotics task simulations, for example flocking or collective foraging. In contrast with the two simulators which were mentioned in the section above, Gazebo does not allow to import 3D meshes and the OpenGL is used for the representation of objects. ARGoS has custom-built physics engine with limited capabilities. For programming methods this application supports scripts written in Lua or C++ languages. What is very advantageous that the scene is saved as XML file and because the other simulators support this format, they can import this file and run a robot simulation. Nowadays, ARGoS is less used, but for small simulation it is excellent.

3.4 Mass properties

Mass can be defined either as a property or a measure of physical body. By the term property we mean any measurable property, which describes the state of an object and the term measure to express the inertia of physical body. When we talk about the mechanics of a complex object it consists of the combination of Newton's laws. In this section we discuss how Newton's [RPFS13] laws influence complicated objects.

3.4.1 Center of Mass

The complicated objects has several kinds as water flowing or galaxies whirling. At the beginning let us define a simple object what is called the rigid body. This rigid body is turning while in motion. The motion can be either simple or complex. Let us take the simplest motion of an object where the body rotates according to a fixed axis. This motion is based at a certain point on a body which moves in a plane perpendicular to this axis. Such rotation is called plane rotation or rotation in two dimensions.

Let us consider the first theorem of motion of complicated objects. This can be better depicted with a particular example. Imagine an object which consists of blocks and spokes which are connected with strings. When you throw it, you observe that there will be an effective center which moves in parabola. That implicates our theorem of the center of mass, which say[RPFS13]: there is a mean position which is mathematically definable, it does not have to be a point of the material itself. So let us consider any object which consists of little particles with different forces among them. Let us define i as index of one particle. Then the force on the i-th particle we can express as follow:

$$F_i = m_i(d^2 r_i/dt^2) \tag{3.1}$$

In circumstances when all parts are moving slower than the speed of light and we use the nonrelativistic approximation, in that case mass is constant and we can express it as[RPFS13]:

$$F_i = d^2(m_i r_i)/dt^2 \tag{3.2}$$

Next, from that we compute the total force **F** by the sum of all forces for all different indexes:

$$\sum_i F_i = F = \frac{d^2(\sum_i m_i r_i)}{dt^2} \tag{3.3}$$

At this state we have an equation by which we compute the total force. Now we need to rewrite this equation as the total mass times some acceleration. Let define **M** which is a total mass. Then the certain vector **R** looks like:

$$R = \sum_i m_i r_i/M \tag{3.4}$$

After that we can simply express

$$F = d^2(MR)/dt^2 = M(d^2 R/dt^2) \tag{3.5}$$

Since M is constant the equation 3.5 is correct. At the end we consider that we can express the external force by the total mass times the acceleration of an imaginary point whose location is **R**. We call this point as center of mass of the rigid body.

3.4.2 Moment of inertia

The moment of inertia or in other words rotational inertia is a quantitative measure that determines the torque needed for rotational motion about rotation axis. Rotational inertia depends on mass and the selected axis, because it is important how the mass is distributed around rotational axis and it will change depending on the selected axis. When we talk about point-like mass the moment of inertia is given by mr^2. In this formula m is the mass and r is the distance from an axis to a point. In other words, rotational inertia for a rigid body is a sum of all the parts of mass multiplied by the square of its distances from the selected axis.

You can find moment of inertia in Newton's laws of motion of a rigid body where its shape and mass are combined. There are some differences when we speak about planar or spatial movements. Moment of inertia in the first case is a scalar and in the second case is a symmetric 3x3 matrix called inertia tensor or inertia matrix.

Finding the rotation inertia for different objects can be a bit problematic. Let us see how to get it. We can express the moment of inertia about the z-axis as follows [RPFS13]:

$$I = \sum_i m_i(x_i^2 + y_i^2) \tag{3.6}$$

or

$$I = \int (x^2 + y^2)dm = \int (x^2 + y^2)pdV \tag{3.7}$$

According to the formula 3.6, we get the inertia when we sum all the masses and then multiply it by the distance $x_i^2 + y_i^2$ from the axis. Even if we talk about the three-dimensional object we have to square only the two-dimensional distance. In three-dimensions the formula for rotation is about the z-axis.

Let us see an example to consider how to compute the moment of inertia for a rod, which is rotating about a perpendicular axis. Now, in this case **y** being zero and it is enough to sum all the masses and multiply it by the x-distances squared.[RPFS13]

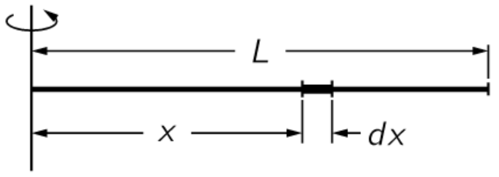


Figure 3.5: Straight rod rotating around an axis adopted from [RPFS13]

Divide the rod into small pieces of length dx. If dx is the total length of the rod and mass will be M, then applies the following:

$$dm = Mdx/L \tag{3.8}$$

After that, we know the sum, which is the integral of x^2 and the dm,then we write:

$$I = \int_0^L x^2 \frac{Mdx}{L} = \frac{M}{L} \int_0^L x^2 dx = \frac{ML^2}{3} \tag{3.9}$$

The moment of inertia is the mass times the length squared, but the important part is the factor 1/3 with the help of the integral. The computation 3.9 will demonstrate when the rotation axis is on the end of the rod. If we want the rotation axis to be in the middle of the rod we have to compute the integral again between the range from $-\frac{1}{2}L$ to $+\frac{1}{2}L$. But we could solve this more easily if we divided the rod into two parts where each the mass will be

M/2 and the length L/2. After that we will write the next formula

$$I = \frac{2(M/2)(L/2)^2}{3} = \frac{ML^2}{12} \tag{3.10}$$

We could compute the moment of inertia for many various bodies. But what is more important, there is a theorem which is interesting at this part. This theorem is called the parallel-axis theorem. Let us describe what this theorem means. Imagine that we have an object and we want to find the moment of inertia needed for the rotation around any axis. We suppose that we support the object on pivots at the center of mass. When we do that and we start moving it, then the object will not rotate around the selected axis. To make a body move we need the same forces as when the mass were concentrated at the center of the mass, so for the rotation inertia we will write

$$I_1 = MR_{CM}^2 \tag{3.11}$$

where R_{CM} or R_{COM} is the distance from the axis to the center of the mass. But we must note, that the formula 3.11 is not a right formula, because not only the center of the body is moving in a circle, but we must turn it around its center of mass. At the end to I_1 we need to add the rotation inertia I_c. So the right formula is supposed to be

$$I = I_c + MR_{CM}^2 \tag{3.12}$$

This is a parallel-axis theorem. Let us see how to do it. The moment of inertia is

$$I = \sum (x_i^2 + y_i^2) m_i$$

We will explain only the x's, but the y's work the same way. We have x, which is the distance of a particular point mass from the origin. Let us take x' from the center of mass and not from the origin. Then we write

$$x_i = x'_i + X_{CM} \tag{3.13}$$

When we square the right side of formula,

$$x_i^2 = x_i'^2 + 2X_{CM}x'_i + X_{CM}^2 \tag{3.14}$$

After that multiply all part by m_i, next summed for all i and move the constants before the summation. We get

$$I_x = \sum m_i x_i'^2 + 2X_{CM} \sum m_i x'_i + X_{CM}^2 \sum m_i \tag{3.15}$$

The formula 3.15 is very easy to understand. The first sum is the x part of I_c. The second sum expresses the total mass which is multiplied with x'-coordinate of the CM.

3.5 Kinematics

Kinematics deals with the motion of bodies and it does not take into account the forces which cause the motion. In the field of robot kinematics, geometry is applied to the movement of kinematic chains which has its own degree of freedom. It is very difficult and important for an industrial manipulator to create the best kinematics model for a robot. In kinematics modelling there are two different spaces. One is the Cartesian space and the second is the Quaternion space. The transformation in Cartesian coordinate system is divided into rotation and translation. For the rotation representation we know many ways as Euler angles, Gibbs vector, Cayley-Klein parameters, Hamilton quaternion's. Denavit and Hartenberg show that for the rotation and translation between two consecutive frames we need four parameters. This convention is called DH convention and these four parameters are called D-H parameters. This convention becomes the standard in the robot kinematics.[KB06]

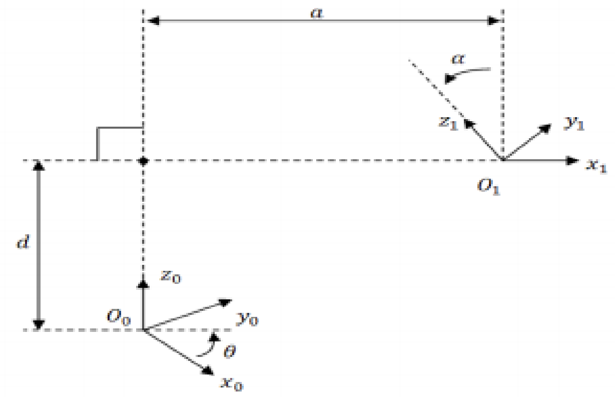


Figure 3.6: Two coordinate frame systems adopted from [TPS17]

D-H parameters [TPS17] are

- **Link length (a_i): this is the distance between z_0 and z_1. We could express it with (Trans,z,a_i)**
- **Link offset (d_i): it is the distance from origin** O_0 to the intersection of axis x_1 with z_0. We could express it with (Trans, z, a_i)
- **Joint angle (Θ_i): angle from** x_0 and x_1 measured in plane normal to z_0. We could express it with (ROT,z,Θ_i)
- **Link twist (α_i): angle between** Z_0 and Z_1, measured in plane normal to X_1 axis. We could express it with (ROT, x, α_i)

The robot kinematics consists of two types, the forward or direct kinematics and inverse kinematics. The first type is easier because there is no deriving the equations. The second is a more difficult problem, because it is computationally expensive thanks to properties of singularity and nonlinearity, and the control of manipulators takes a long time. What is very useful in forward kinematics is that you always find a kinematic solution for manipulators. You can see the relationship between forward and inverse kinematics in the picture below

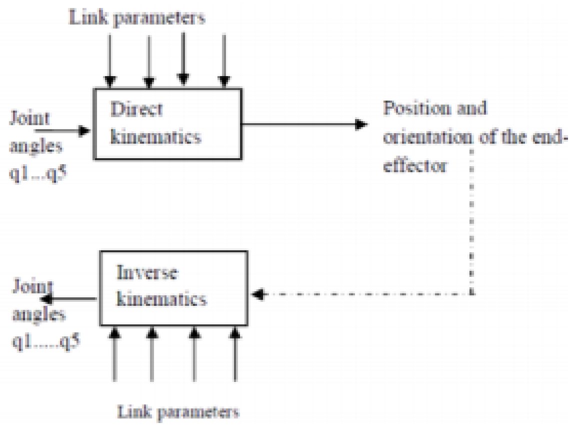


Figure 3.7: Relationship between forward and inverse kinematics adopted from [TPS17]

To solve the inverse kinematics problems we know two main methods numerical and analytical. At the numerical method we get the joint variables from numerical solution and at the analytical we follow the configuration data. Next, there are two approaches for the analytical solution: geometric and algebraic. The geometric solution is applied to the simplest structures and to complicated structures as arm extended into three dimension.

3.5.1 Forward kinematics

If we know the values for the joint variables we can determine the position and orientation of end-effector. When we talk about revolute or rotational joint, the joint variables are angles between the links and in the case of prismatic and sliding joints the variables are angles between the link extension.

Let us take a robot with n joints. This robot has n+1 links if each joint connects two links. Then we define the joints from 1 to n and links from 0 to n. In this order the link with number 0 is the base link and the i-th joint connects link i-1 to link i. The base link is fixed and link i moves, when joint i is actuated. Now for every joint has to be associated a joint variable. Denote it with q_i. The meaning of the joint variable depends on the type of joint. [Don]

$$q_i = \begin{cases} \theta_i & : \text{joint i revolute} \\ d_i & : \text{joint i prismatic} \end{cases}$$

Figure 3.8: Joint variable meaning for revolute and prismatic joint adopted from [Don]

When we speak about revolute joint then the joint variable is the angle of rotation and in terms of prismatic joint the joint variable is the joint displacement. To show the kinematic analysis, in the picture below we add a coordinate frame to each link.[Don]

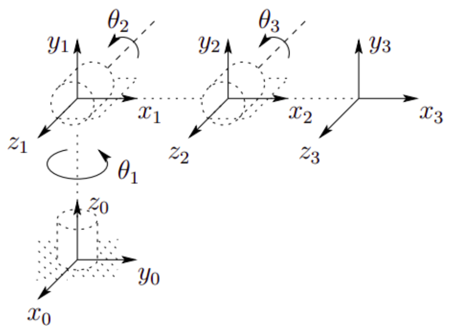


Figure 3.9: Coordinate frames added to elbow links adopted from [Don]

To the link i we attach $o_i x_i y_i z_i$ [Don]. When a robot executes whatever motion, each point on the link **i** is constant in case when coordinates are expressed in the i-th coordinate frame. Moreover the actuation of joint i has an impact on its attached frame $(o_i x_i y_i z_i)$. The robot base we call the inertial frame and we denote it $o_0 x_0 y_0 z_0$.

When we have the coordinate frames, we suppose the homogeneous transformation matrix A_i which expresses the position and orientation of $o_i x_i y_i z_i$ regarding the link i - 1 frame, that means the $o_{i-1} x_{i-1} y_{i-1} z_{i-1}$ [Don]. The transformation matrix changes according to the configuration of the robot. We can deduce, then A_i exists as a function of a single joint variable, which means

$$A_i = A_i(q_i) \tag{3.16}$$

We already know that the homogeneous transformation matrix which expresses the pose of $o_j x_j y_j z_j$ regarding $o_i x_i y_i z_i$ is called a **transformation matrix**[Don], and it is labeled as T_j^i. Let us introduce some equations which apply

$$T_j^i = A_{i+1}A_{i+2}...A_{j-1}A_j \quad if \quad i < j \tag{3.17}$$

$$T_j^i = I \quad if \quad i = j \tag{3.18}$$

$$T_j^i = (T_i^j)^-1 \quad if \quad j > i \tag{3.19}$$

We have added the coordinate frame to all the links, henceforward any point on the end-effector will be constant, independent from a robot configuration. Pose of the end-effector regarding the base frame is expressed by a three-vector O_n^0 and with the 3x3 rotation matrix R_n^0 [Don]. Using these relationships we can define the homogeneous transformation matrix

$$H = \begin{bmatrix} R_n^0 & O_n^0 \\ 0 & 1 \end{bmatrix} \tag{3.20}$$

Then the pose of the end-effector in the base frame is the next

$$H = T_n^0 = A_1(q_1)...A_n(q_n) \tag{3.21}$$

Then each homogeneous transformation matrix has the following form

$$A_i = \begin{bmatrix} R_i^{i-1} & O_i^{i-1} \\ 0 & 1 \end{bmatrix} \tag{3.22}$$

So

$$T_j^i = A_{i+1}...A_j = \begin{bmatrix} R_j^i & O_j^i \\ 0 & 1 \end{bmatrix} \tag{3.23}$$

The matrix R_j^i is the orientation of $o_j x_j y_j z_j$ relative to $o_i x_i y_i z_i$. It describes the rotational part of the homogeneous matrices. We will write it as follows [Don]

$$R_j^i = R_{i+1}^i ... R_j^{j-1} \tag{3.24}$$

We can write the coordinate vectors O_j^i recursively as

$$O_j^i = O_{j-1}^i + R_{j-1}^i O_j^{j-1} \tag{3.25}$$

This is the forward kinematics. But there exists a better solution which contains simplifications. That convention is called the Denavit-Hartenberg representation.

In the previous solution we attached a coordinate frame for all links, but it would be good to be systematic. That means we need to be careful about the frame choices. In kinematics introduction we introduced the mostly used convention for selecting frames. This could be the Denavit-Hartenberg or D-H convention. This convention represents each homogeneous transformation

A_i as a product of four transformations. [Don]

$$\begin{aligned}
A_i &= R_{z,\Theta_i} Trans_{z,d_i} Trans_{x,a_i} R_{x,\alpha_i} \\
&= \begin{bmatrix} c\theta_i & -s\theta_i & 0 & 0 \\ s\theta_i & c\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c\alpha_i & -s\alpha_i & 0 \\ 0 & s\alpha_i & c\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} c\theta_i & -s\theta_i c\alpha_i & s\theta_i s\alpha_i & a_i c\theta_i \\ s\theta_i & c\theta_i c\alpha_i & -c\theta_i s\alpha_i & a_i s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{3.26}$$

As we noticed at the beginning the θ_i, a_i, d_i, α_i are the D-H parameters and these are called joint angle, link length, link offset and link twist. The matrix A_i is a function of a single variable which is the joint variable.

3.5.2 Inverse kinematics

The inverse kinematics is needed in the control of robot manipulators. Computationally it is very expensive and it takes a long time to control the manipulators. Tasks are in the Cartesian space and actuators are in the joint space. In the Cartesian space we have an orientation matrix and position vector while the joint space is represented by the joint angles. The conversion between the Cartesian space and the joint space is called the inverse kinematics problem. There are two types of solutions: the geometric and the algebraic. Let us see the algebraic solution.[KB06]

3.5.2.1 Geometric solution

The geometric solution breaks down the spatial geometry of the robot manipulators into several plane geometries. Let us have a simple structure like planer manipulator which has two degree of freedom. The joints are both revolute which connect the links. The lengths are expressed as l_1 and l_2 [KB06].

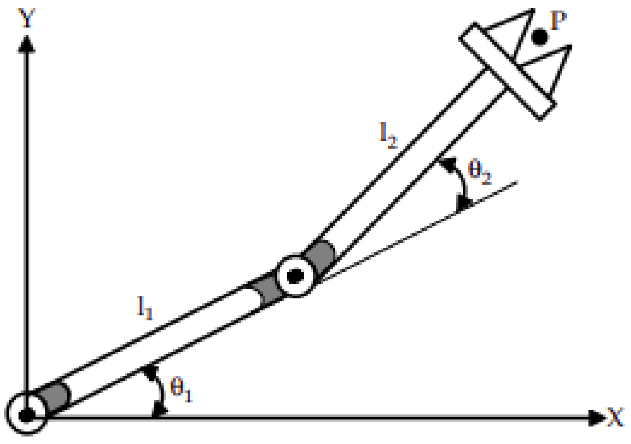


Figure 3.10: Planar manipulator with 2-DOF adopted from [KB06]

In the next picture we can see the derived kinematics equations for the planar manipulator.

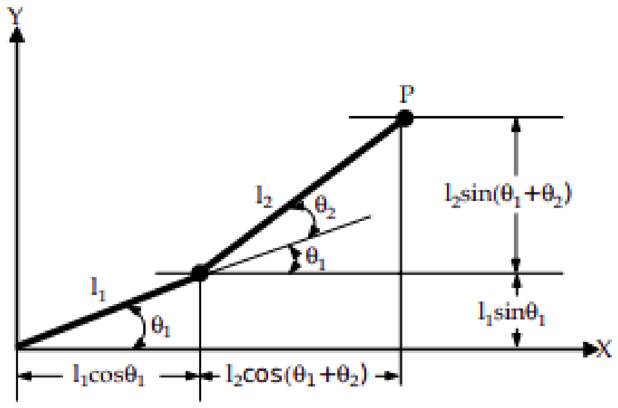


Figure 3.11: Derived equations for a planar manipulator adopted from [KB06]

The components p_x and p_y of the point P are expressed as

$$p_x = l_1 c\theta_1 + l_2 c\theta_{12} \tag{3.27}$$

$$p_y = l_1 s\theta_1 + l_2 s\theta_{12} \tag{3.28}$$

The expression [KB06] $c\theta_{12} = c\theta_1 c\theta_2 - s\theta_1 s\theta_2$ and $s\theta_{12} = s\theta_1 c\theta_2 + c\theta_1 s\theta_2$. The θ_2 we can compute from the summation of equations 3.27 and 3.28.

$p_x^2 = l_1^2 c^2\theta_1 + l_2^2 c^2\theta_{12} + 2l_1 l_2 c\theta_1 c\theta_{12}$

$p_y^2 = l_1^2 s^2\theta_1 + l_2^2 s^2\theta_{12} + 2l_1 l_2 s\theta_1 s\theta_{12}$

$p_x^2 + p_y^2 = l_1^2(c^2\theta_1 + s^2\theta_1) + l_2^2(c^2\theta_{12} + s^2\theta_{12}) + 2l_1 l_2(c\theta_1 c\theta_{12} + s\theta_1 s\theta_{12})$

When we know $c^2\theta_1 + s^2\theta_1 = 1$ then we can simplify the equation like

$p_x^2 + p_y^2 = l_1^2 + l_2^2 + 2l_1 l_2(c\theta_1[c\theta_1 c\theta_2 - s\theta_1 s\theta_2] + s\theta_1[s\theta_1 c\theta_2 + c\theta_1 s\theta_2])$

$p_x^2 + p_y^2 = l_1^2 + l_2^2 + 2l_1 l_2(c^2\theta_1 c\theta_2 - c\theta_1 s\theta_1 s\theta_2 + s^2\theta_1 c\theta_2 + c\theta_1 s\theta_1 s\theta_2)$

$p_x^2 + p_y^2 = l_1^2 + l_2^2 + 2l_1 l_2(c\theta_2[c^2\theta_1 + s^2\theta_1])$

$p_x^2 + p_y^2 = l_1^2 + l_2^2 + 2l_1 l_2 c\theta_2$

We can express from this $c\theta_2$ like

$$c\theta_2 = \frac{p_x^2 + p_y^2 - l_1^2 - l_2^2}{2l_1l_2} \tag{3.29}$$

Since, $c^2\theta_i + s^2\theta_i = 1$ where $i = 1, 2, 3, ...$, $s\theta_2$ is expressed as

$$s\theta_2 = {}^{+-}\sqrt{1 - (\frac{p_x^2 + p_y^2 - l_1^2 - l_2^2}{2l_1l_2})^2} \tag{3.30}$$

At last, the solutions for the θ_2 can be written as

$$\theta_2 = \text{atan2}({}^{+-}\sqrt{1 - (\frac{p_x^2 + p_y^2 - l_1^2 - l_2^2}{2l_1l_2})^2}, \frac{p_x^2 + p_y^2 - l_1^2 - l_2^2}{2l_1l_2}) \tag{3.31}$$

Let us find the solution of θ_1. Now we known θ_2 and we need to multiply the equation 3.27 by $c\theta_1$ and equation 3.28 by $s\theta_1$. So we get [KB06]

$c\theta_1 p_x = l_1 c^2\theta_1 + l_2 c^2\theta_1 c\theta_2 - l_2 c\theta_1 s\theta_1 s\theta_2$

$s\theta_1 p_y = l_1 s^2\theta_1 + l_2 s^2\theta_1 c\theta_2 + l_2 c\theta_2(c^2\theta_1 + s^2\theta_1)$

$c\theta_1 p_x + s\theta_1 p_y = l_1(c^2\theta_1 + s^2\theta_1) + l_2 c\theta_2(c^2\theta_1 + s^2\theta_1)$

We can simplify the equation as

$$c\theta_1 p_x + s\theta_1 p_y = l_1 + l_2 c\theta_2 \tag{3.32}$$

In the next step, we multiply the equation 3.27 by $-s\theta_1$ and equation 3.28 by $c\theta_1$

$-s\theta_1 p_x = -l_1 s\theta_1 c\theta_1 - l_2 s\theta_1 c\theta_1 c\theta_2 + l_2 s^2\theta_1 s\theta_2$

$c\theta_1 p_y = l_1 s\theta_1 c\theta_1 + l_2 c\theta_1 s\theta_1 c\theta_2 + l_2 c^2\theta_1 s\theta_2$

$-s\theta_1 p_x + c\theta_1 p_x = l_2 s\theta_2(c^2\theta_1 + s^2\theta_1)$

After the simplification we get the expression

$$-s\theta_1 p_x + c\theta_1 p_y = l_2 s\theta_2 \tag{3.33}$$

Now, we need to multiply equation 3.32 by p_x and equation 3.33 by p_y and after that sum these equations to get $c\theta_1$ [KB06].

$c\theta_1 p_x^2 + s\theta_1 p_x p_y = p_x(l_1 + l_2 c\theta_2)$

$-s\theta_1 p_x p_y + c\theta_1 p_y^2 = p_y l_2 s\theta_2$

$c\theta_1(p_x^2 + p_y^2) = p_x(l_1 + l_2 c\theta_2) + p_y l_2 s\theta_2$

When we adjust the last equation, then we get

$$c\theta_1 = \frac{p_x(l_1 + l_2 c\theta_2) + p_y l_2 s\theta_2}{p_x^2 + p_y^2} \tag{3.34}$$

So, $s\theta_1$ is given by

$$s\theta_1 = {}^{+}_{-}\sqrt{1 - (\frac{p_x(l_1 + l_2 c\theta_2) + p_y l_2 s\theta_2}{p_x^2 + p_y^2})^2} \tag{3.35}$$

At last, the solutions for the θ_1 can be written as

$$\theta_1 = \text{atan2}({}^{+}_{-}\sqrt{1 - (\frac{p_x(l_1 + l_2 c\theta_2) + p_y l_2 s\theta_2}{p_x^2 + p_y^2})^2}, \frac{p_x(l_1 + l_2 c\theta_2) + p_y l_2 s\theta_2}{p_x^2 + p_y^2}) \tag{3.36}$$

While we have above the computation for a very simple structure, as can you seen it was not so easy.

3.5.2.2 Algebraic solution

If we have a robot manipulator which is more difficult, for example an arm which extends into three dimensions the geometry will be more complex.

To deal with this problem the algebraic approach is the best way. Let us consider a robot manipulator with six joints and the pose of its end-effector regarding base is given by [KB06]

$$T_6^0 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = T_1^0(q_1)T_2^1(q_2)T_3^2(q_3)T_4^3(q_4)T_5^4(q_5)T_6^5(q_6)$$

So we can find a solution for joint q_1 as a function of the T transformation matrix. We will use the link transformation inverses as multiplication.

$$[T_1^0(q_1)]^{-1}T_6^0 = [T_1^0(q_1)]^{-1}T_1^0(q_1)T_2^1(q_2)T_3^2(q_3)T_4^3(q_4)T_5^4(q_5)T_6^5(q_6)$$

$[T_1^0(q_1)]^{-1}T_1^0(q_1) = I$ where I is an identity matrix. Taking this into account, the above shown equation will be changed to [KB06]

$$[T_1^0(q_1)]^{-1}T_6^0 = T_2^1(q_2)T_3^2(q_3)T_4^3(q_4)T_5^4(q_5)T_6^5(q_6)$$

We will find the other joint variables with the same method as

$$[T_1^0(q_1)T_2^1(q_2)]^{-1}T_6^0 = T_3^2(q_3)T_4^3(q_4)T_5^4(q_5)T_6^5(q_6)$$

$$[T_1^0(q_1)T_2^1(q_2)T_3^2(q_3)]^{-1}T_6^0 = T_4^3(q_4)T_5^4(q_5)T_6^5(q_6)$$

$$[T_1^0(q_1)T_2^1(q_2)T_3^2(q_3)T_4^3(q_4)]^{-1}T_6^0 = T_5^4(q_5)T_6^5(q_6)$$

$$[T_1^0(q_1)T_2^1(q_2)T_3^2(q_3)T_4^3(q_4)T_5^4(q_5)]^{-1}T_6^0 = T_6^5(q_6)$$

If the elements on the left side of the equation, that are functions q1, we calculate using the right side element, then q_1 can be computed as functions of r_{11}, r_{12}, ... r_{33}, p_x, p_y, p_z and the fixed link parameters. If q1 is solved, then joint variables can be solved similarly q_2, q_3, q_4, q_5, q_6. [KB06]

Chapter 4

Design of robot Lilli model

In this section we introduce how our robot was designed, what is the best way to build the clean model which is a base structure for the simulator. Next we show the right shapes, joint types and usages, how the dynamic shapes would look like, what is a model definition and how to design the URDF format for our robot too. In the second part of the design we are going to describe the simulator, appearance of the scene, the definition of the model in the scene, robot control via remote API, inverse kinematics solver and its function, IK and FK solving and calculation methods. At the end, we acquire the best practices how to build the clean model not only for our robot, but also for others. We give the reader an overview how to use inverse kinematics solver and methods for the simulation.

4.1 Clean model

This section describes the process of building the clean model, which we will use for a simulation. Building a clean model, of a robot or any robot manipulator is very important. We give the clean model not only good looks, but what is more important is fast displaying, fast simulating and the main reason is the stable model. In the next subsections we describe which shapes are the best choice for the model, the joint definition, dynamic shapes if we want the robot to be dynamically enabled and the model definition. Of course, we have chosen the CoppeliaSim (V-REP) simulator for the simulation purpose and we need to pay attention to the convention between the simulator and the model. For this reason our model must contain the correct shapes, joints and it has to have the right model definition. Of course we have a few choices to define the clean model and at the end of this section we will know what they are.

4.1.1 Shapes

When we build a new model, then the first thing is to handle its visual aspect. In CoppeliaSim we either create primitive shapes directly or our second choice is to import a mesh from an external application. When we do this through the simulator then we have a choice to select from pure shapes or regular shapes. Pure shape is generally optimized for dynamic interaction and it is dynamically enabled.

When importing CAD shape or model to the simulator, then we need to check if the model is not too heavy. Heavy means that it contains too many triangles and it slows down the calculations.

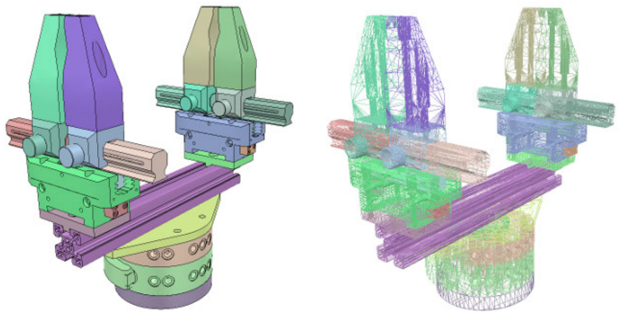


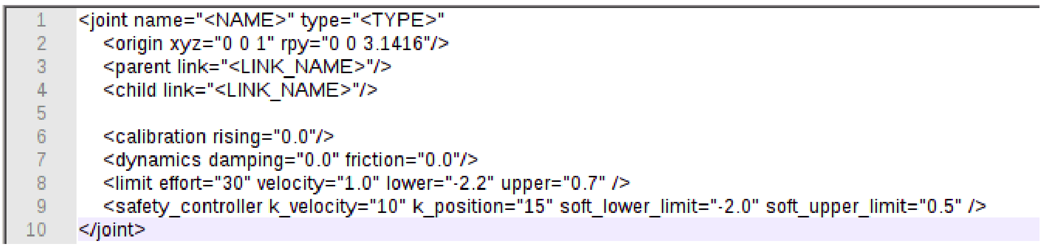
Figure 4.1: A complex CAD model adopted from [Copa]

In the picture you see a very heavy model which contains many triangles and when you need this model to interact with another robots or parts or with the environment then the scene becomes too slow and simulation calculations too. Generally, the recommended total count of triangles for the robot manipulator is maximum 20 000. What you need to pay attention to is how your shape is built. You need to exclude from the shape model as few as possible holes, small details, because it requires a lot of triangular faces, and you need to exclude also the inside of objects. The last important thing is set the level of details while the mesh is exported from CAD system. Firstly, export the large objects with adjusted precision and after that the small objects with a set up precision.

Before you start modeling your shapes, it is good to know which CAD data is supported by CoppeliaSim. These formats are the following: STL, DXF, OBJ and Collada. There are two other types which support CoppeliaSim. These formats are SDF and URDF format. These are not pure mesh-based file formats, but both have their advantages to describe the robot manipulator.

4.1.2 Joints

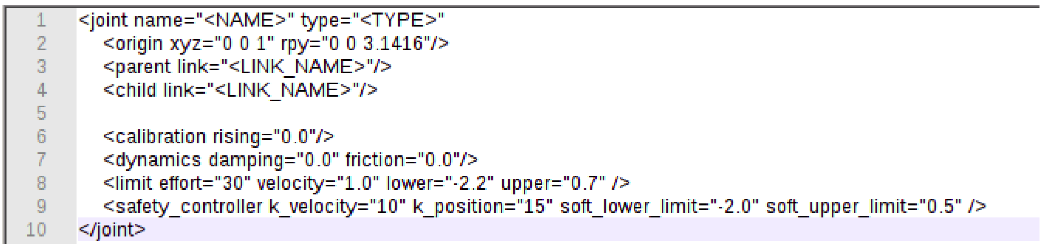
When we know, how to model our shapes, then the next important part is joints. For defining the robot joints we have more ways to do so. CoppeliaSim allows to define our joint position and orientation directly in the simulator. This can be done in two ways. First, if we know the position and orientation of joints then with a dialog window we simply pose it. The second, when we have only the Denavit-Hartenberg parameters. For this purpose CoppeliaSim offers the possibility to build the joints via tool model which is located in the model browser. In case we do not know the position and orientation of the joints we can extract them from the imported meshes. The next way is to define the joint in URDF or SDF file. In this case the joint definition is similar as we use the <Joint> xml element which describes the kinematics and dynamics of the joint. This element has many properties which describe the joint properties.



```
<joint name="<NAME>" type="<TYPE>"
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="<LINK_NAME>"/>
   <child link="<LINK_NAME>"/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
   <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_limit="0.5" />
</joint>
```

Figure 4.2: <Joint> element for describing the joint in URDF

And in SDF this looks like



```
<joint name="<NAME>" type="<TYPE>"
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="<LINK_NAME>"/>
   <child link="<LINK_NAME>"/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
   <safety_controller k_velocity="10" k_position="15" soft_lower_limit="-2.0" soft_upper_limit="0.5" />
</joint>
```

Figure 4.3: <Joint> element in SDF

Next in CoppeliaSim a joint has two reference frames. The first frame is fixed and the second is not. The second frame will move relatively to the first frame depending on the joint configuration. CoppeliaSim supports four joint types which are revolute, prismatic, spherical joint and screws.

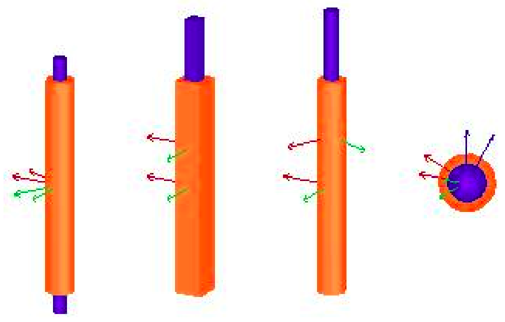


Figure 4.4: Four joint types in order revolute, prismatic, screws and spherical joint adopted from [Copb]

- Revolute joint - connects two links which have 1 DOF and it is used for rotational movement. This movement has its upper and lower limits. Usually the value is defined in radians.
- Prismatic joint - connects two links which have one DOF and it is used for translational movement. It has an upper and lower limit range.
- Spherical joint - this joint realizes multi DOF motion, it has three values which describe the rotation around the first reference frame. These values are the Euler angles.
- Screws - it is a combination of revolute and prismatic joint. It has one DOF and a movement similar to the screw.

CoppeliaSim enables some modes for the joints. These modes are passive, inverse kinematics, dependent, motion and torque or force mode. When a joint is in the passive mode it behaves as a fixed link. In inverse kinematics mode a joint acts according to inverse kinematics calculations. In the dependent mode a joint is linked with another joint by linear equation. The last mode, torque or force mode is set when the joint is simulated by the selected physics engine.

4.1.3 Dynamic shapes

We want shapes to be dynamically enabled, but some cases we need to configure manually. This means our robot will fall, react to collisions etc.. Let us see what properties the shape has. Shape can be **dynamic or static** and **respondable or non-respondable**. Dynamic shape falls out and it is affected by forces, till the static shape stays in place or follows the parent movements. The second property respondable is responsible for a collision reaction when two respondable shapes collide. If the shape is non-respondable, then shapes do not compute the collision response.

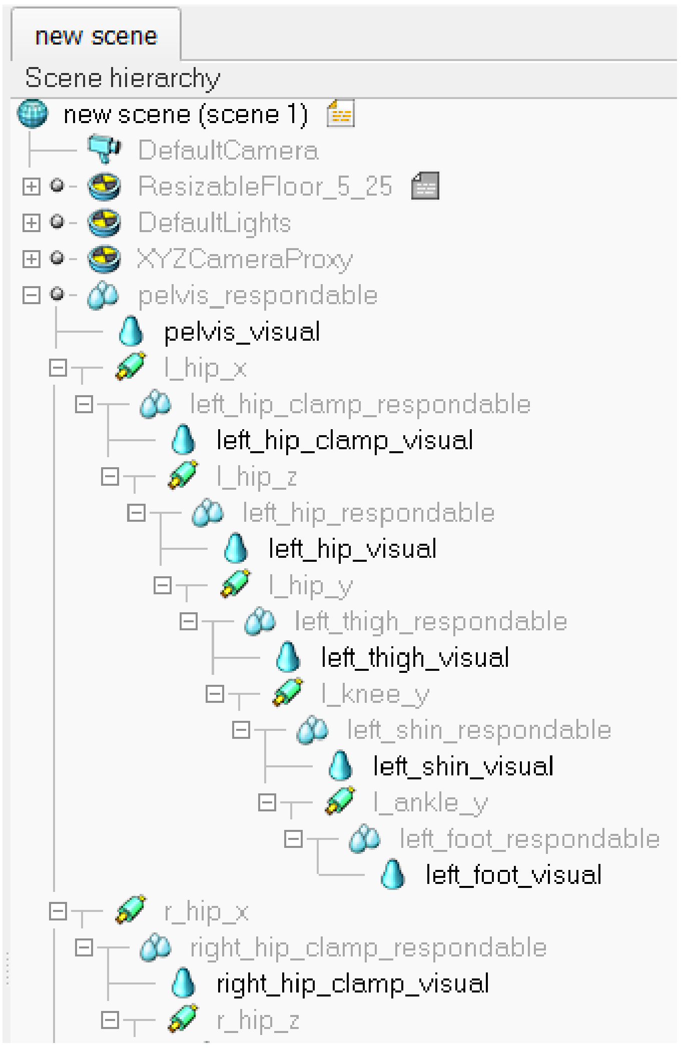
While building a clean model, we have to be careful which types of shapes can physics engine simulate. There are 5 types:

- Pure shapes - these are stable and efficiently handled by the physics engine, but they are limited in geometry.
- Pure compound shapes - these are groups of several pure shapes and the same goes for it regarding the properties.
- Convex shapes - convex shapes are less stable and the physics engine handle these shapes a little bit worse than pure shapes.
- Convex decomposed shapes - these are groups of convex shapes and their properties are similar to convex shapes

- Random shapes - random shapes have a poor performance and it is not recommended to use these shapes in CoppeliaSim

4.1.4 Model definition

Our selected robot simulator has its own model definition. The model is a part of the scene and exists only in the "*.ttm" file. We understand this model as a group of scene objects built according to the tree structure. This model always has a base part which is called a model base. Various objects need to connect to the model base to create the right model. The right model looks like as follows



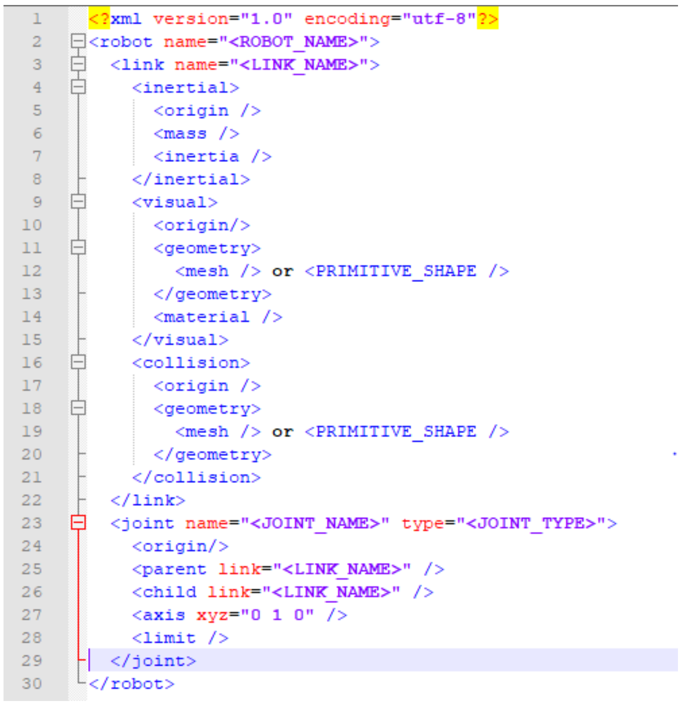
```
new scene
Scene hierarchy
new scene (scene 1)
  DefaultCamera
  ResizableFloor_5_25
  DefaultLights
  XYZCameraProxy
  pelvis_respondable
    pelvis_visual
    l_hip_x
      left_hip_clamp_respondable
        left_hip_clamp_visual
        l_hip_z
          left_hip_respondable
            left_hip_visual
            l_hip_y
              left_thigh_respondable
                left_thigh_visual
                l_knee_y
                  left_shin_respondable
                    left_shin_visual
                    l_ankle_y
                      left_foot_respondable
                        left_foot_visual
    r_hip_x
      right_hip_clamp_respondable
        right_hip_clamp_visual
        r_hip_z
```

Figure 4.5: Scene hierarchy

If you need, you will protect the model, which does not allow manipulation by individual objects. Except of this you can do more settings to manipulate with the model, but at this point it is enough to know the above mentioned facts.

4.1.5 URDF

We introduce important parts of the clean model as the manipulator shapes, joints and the dynamic property of shapes. At this point we know how various shapes can be modeled for the best performance at the simulator, how to select or define joints instead of servos, the dynamically enabled shapes for simulation purpose and how the model definition looks like in our selected simulator. As the next step, we must clear how to get the robot manipulator into the simulator, where all parts of the robot are modeled via CAD system. Indeed, we have the option to import all CAD parts separately, but then we need to create the tree structure inside the simulator. It is not the best way because when we need to use the robot manipulator in another simulator, then we need to build all tree structure again. For this purpose we selected the URDF file format, which gives us a very nice structure to describe the robot links, joints and their relationship. URDF uses the XML format to describe all elements of a robot. With this description format we are able to describe simple and complex robot manipulator structures. Let us see how URDF builds a complete robot manipulator. At first we define the <robot> element which has name property. Inside the robot element you must define the links and joints. The model has a tree structure and we can deduce that we have parent-child relationships between joints and links. That means a joint connects two links and it cannot happen that another link is connected to this link. There exists a way to define this relation, but you need to define instead of a link a fixed joint type and manipulate with the layers in the simulator. Furthermore, you need to position another visible object to the joint position. Usually the first link is the model base object. Next is the child link and after it the joint, which connects the pairs. In the picture below you can see which elements are usually required to describe a robot.



```xml
<?xml version="1.0" encoding="utf-8"?>
<robot name="<ROBOT_NAME>">
  <link name="<LINK_NAME>">
    <inertial>
      <origin />
      <mass />
      <inertia />
    </inertial>
    <visual>
      <origin/>
      <geometry>
        <mesh /> or <PRIMITIVE_SHAPE />
      </geometry>
      <material />
    </visual>
    <collision>
      <origin />
      <geometry>
        <mesh /> or <PRIMITIVE_SHAPE />
      </geometry>
    </collision>
  </link>
  <joint name="<JOINT_NAME>" type="<JOINT_TYPE>">
    <origin/>
    <parent link="<LINK_NAME>" />
    <child link="<LINK_NAME>" />
    <axis xyz="0 1 0" />
    <limit />
  </joint>
</robot>
```

Figure 4.6: one link and one joint definition in URDF

The link element has three important tags. First is the tag <inertial> which includes the center of mass of the link, the mass and 6 above-diagonal elements from the moment of inertia matrix. The next two tags the <visual> and <collision> tags have almost the same inner elements. They both have an origin definition and geometry. Geometry describes the link visual appearance by mesh or some primitive shapes like cube or cylinder etc.. When you describe the joint you must define its name, parent, child, the rotation axis and the joint limits. Joint limits contain the maximum force supported by a joint, upper and lower angle limits and the velocity, which enforces the maximum joint velocity.

4.2 The simulator

According to the previous section, we know how to build the clean model and how to wrap it to the sake of the simulator. In this section we introduce the simulator itself. Every simulator has its own characteristic, but we need to focus on our selected simulator which is CoppeliaSim. We show how the scene looks like and what elements are specific to it, next we describe the simulation script and the main script. After that we show the programming language and external application for communication with the simulations, the calculation modules like the inverse kinematics module. We introduce how to define inverse kinematics groups and elements and which methods are used for the calculation itself.

4.2.1 Scenes

The scene contains the same type of elements as the models, but the scene has some specific elements which are the following: the main script, the environment, pages and views.

The main script serves as the simulation script. Each scene usually has one main script. The main script has its own structure and functions, which allow a simulation to run. Typically the main script has four functions by default, which are called by the system. These function are:

- `sysCall_init` - this function block is executed at the start of the simulation. At this block usually we write the object handles, variables, the shape colors etc..
- `sysCall_actuation` - this function block is responsible for the actuation functionality as inverse kinematics or dynamics etc. and will be executed in each simulation process.

- `sysCall_sensing` - as the function name tells, this function block is responsible for the sensing functionality and it is executed in each simulation process.
- `sysCall_cleanup` - the cleanup function is called before the simulation ends and usually restores the initial configuration.

The environment properties are the following: ambient light, background colors etc.. These properties are not scene objects, but they are part of the scene. This parameters are only saved when the scene is saved.

The difference between pages and views is that the page is the main viewing surface and page may contain one or more views. The view is displaying specific objects as the robot model or camera object and others. It can have floating or fixed position on the page. Each scene has eight pages which we will configure in several ways.

The default scene consists of the environment, the main script, pages and views as many as needed, camera objects, light objects and the floor.

4.2.2 Scripts and Remote API

Controlling the robot manipulator during the simulation CoppeliaSim allows us many possibilities. Simulator allows users to customize every aspect of the simulation by an integrated script interpreter. The language which is used to the scripting is called Lua. This language supports common procedural programming. In the following example we show a simple threaded child script

```
function sysCall_threadmain()
    jointHandles={-1,-1,-1}
    for i=1,3,1 do
```

```
        jointHandles[i]=sim.getObjectHandle('joint_'..i)
    end

    -- RML vectors:
    v=90;a=40;j=80
    currentVelocity={0,0,0,0,0,0,0}
    currentAcceleration={0,0,0,0,0,0,0}
    maxVel={v*math.pi/180,v*math.pi/180,v*math.pi/180}
    maxAccel={a*math.pi/180,a*math.pi/180,a*math.pi/180}
    maxJerk={j*math.pi/180,j*math.pi/180,j*math.pi/180}
    targetVel={0,0,0}

    targetPos1={90*math.pi/180,90*math.pi/180,135*math.pi/180}
    sim.rmlMoveToJointPositions(jointHandles,-1,currentVelocity,
    currentAcceleration,maxVel,maxAccel,maxJerk,targetPos1,targetVel)

    targetPos2={-90*math.pi/180,90*math.pi/180,135*math.pi/180}
    sim.rmlMoveToJointPositions(jointHandles,-1,currentVelocity,
    currentAcceleration,maxVel,maxAccel,maxJerk,targetPos2,targetVel)

    targetPos3={0,0,0}
    sim.rmlMoveToJointPositions(jointHandles,-1,currentVelocity,
    currentAcceleration,maxVel,maxAccel,maxJerk,targetPos3,targetVel)
end
```

This threaded child script is a simple example how to move joints into the target position three times. At the beginning we handle all joint objects from the scene. Next we set variables like velocity, acceleration and jerk and define the RML (Reflexxes motion library) vectors. At the end we set the joint three times to positions targetPos1, tergetPos2, targetPos3. The rmlMoveToJointPositions is a regular API function, which moves more joints at the same time using the RML. The above shown example is a child script. The child script is a simulation script and it represents a particular function in the simulation. The child script is of two different types

- Non-threaded child scripts - include a group of blocking functions, which perform some tasks and after then return control. In case the simulation is halting, then the control is not returned. This type of child script is called from the main script from `sysCall_actuation` and `sysCall_sensing`.
- Threaded child scripts - these scripts are run in a thread. The threaded child script is launched from the main script code. It has some disadvantages like reaction to the simulation stop instruction.

We can be control the model not only from embedded scripts like Lua, but there exists a remote API client that allows to control the robot from an external application using remote API commands. The API commands interact with the simulator through socket communications. The remote API function will be called from various programs as Java, Python, Matlab etc. and allows interaction with the simulator in synchronous and asynchronous mode. If you need to manipulate with the simulator via remote API you must enable it on the client side and server side too. There exist other possibilities, plugins etc. to control a robot, but the above mentioned two types are sufficient for us.

4.2.3 Calculation Modules

CoppeliaSim allows powerful calculation modules, which operate on one or more objects. These modules are the following

- dynamics module - this module allows to simulate objects dynamically
- minimum distance calculation module - this module measures, records and displays the minimum distances between measurable objects.
- collision detection module - this module measures, records and displays the collisions between collidable objects.
- inverse kinematics calculation module - this module solves the IK or FK problems.

Some of the above mentioned calculation methods allow the user to define calculation objects. These objects are linked to the scene objects by operating on them. Calculation objects rely on objects they are connected with as collision object relies on collidable object or IK group relies on dummies where joints have the central role.

4.2.4 Inverse kinematics

The inverse kinematics solver of CoppeliaSim simulator is very flexible. It allows to solve any type of mechanism in forward or inverse kinematics mode. As we have already mentioned in the research, the inverse kinematics problem is finding the joint values, which are corresponding to the position and orientation of an end-effector. In other words we need to find the transformation from the Cartesian space to the joint space. The opposite problem, when we have the joint parameters and we need to find the position of the end-effector is the forward kinematics problem. This can be easier than the IK problem. CoppeliaSim offers recorded results of the IK calculation. For this purpose the simulator uses graph objects.

4.2.4.1 IK groups and IK elements

Our simulator uses the IK groups and IK elements to solve the inverse kinematics problem. In order to be able to use it at 100 percent we need to understand how inverse kinematics is solved. To solve the kinematic chain, IK group must contain at least one or more IK elements. This group contains the solving properties as the solving algorithm, maximum iteration count, etc.. The kinematic chain, which we mentioned before, is specified by the IK element. One kinematic chain represents one IK element and the kinematic chain must contain at least one joint. The IK element consist of

- base object - is the first object at the kinematic chain. It can be objects of any type or a rigid joint. You will select the base object at the dialog window while creating the IK element.
- some links - you will choose from the model links. The joints, which are not in IK mode behave as rigid joints.
- some joints - when a joint has not set the IK mode, then it is considered as a link not a joint.
- a tip - the tip behaves like the end-effector and it is the last object in the kinematic chain. The tip is represented by a dummy object and it is linked with the target dummy.
- a target - the target object is always a dummy. The tip dummy follows the target dummy during the simulation. This means the tip adapts

the target position and orientation when it is available. The tip and target dummies form tip-target pair.

The IK elements are described by a kinematic chain. The base or first object, then several links and joints and at the end the tooltip or sometimes also called the end-effector. In the picture below, we can see a 3 DOF robot manipulator with a base-tip pair

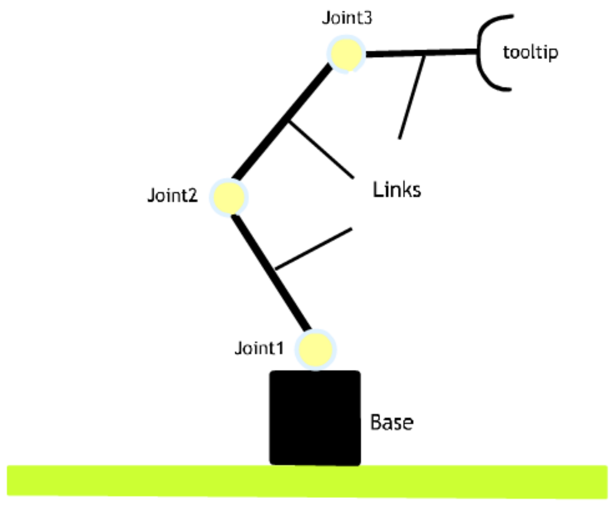


Figure 4.7: Kinematic chain describes the IK element

Next we have a possibility to set the behaviour during the simulation of the kinematic chain. Usually, you want the end-effector to follow the target. Now, if we start the simulation with some additional parameters, the solver calculates the joint variables and the tip moves to the target direction. It is a trivial task. If we have two separate kinematic chains, we need two IK groups to handle the chains at the same time. In this case the IK groups order is omissible. A little tougher case is when the IK element is set on the top of another IK element but they do not share a common joint. It can be seen in Figure 4.8.

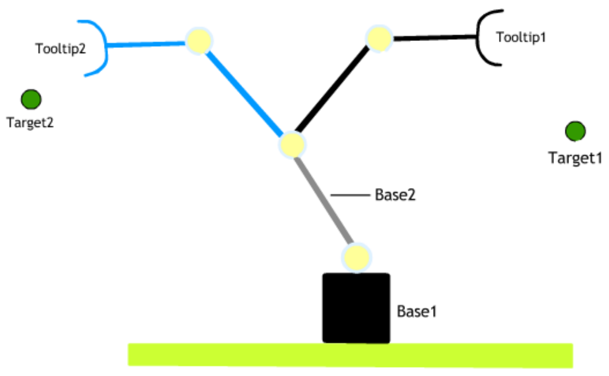


Figure 4.8: Two kinematic chains share common link

The common shared link is the base2 link. The IK group1 task is the Tooltip1-Target1 and the IK group2 is the Tooltip2-Target1 kinematic problem. When the IK solver is solving the Tooltip2-Target2 pair, then the the Base2 link stays in the same position, but when the solver is solving the Tooltip1-Target1 pair, then it displaces the Base2 link. Because of that we must set a sequential solving which means we must solve the IK group1 task before the IK group2 task.

Next difficult case is when kinematic chains share common joints. In this case we are simultaneously solving instead of sequential solving. When this situation happens, then we need group IK elements to one common IK group.

4.2.4.2 Solving IK and FK

The inverse kinematics and forward kinematics use the IK groups and IK elements to solve kinematic tasks. To successfully configure the calculation we need to check some settings. These are the following

- defining for the model a base and tip object (first/last)

- defining the target dummy, whose position and orientation will be followed by the end-effector
- set the IK tip-target pair at tree model
- create and group the IK elements into IK group(s)
- order the IK groups according to the kinematic chains behaviour
- enable inverse kinematic mode in the joint dialog window
- check the IK elements are not overconstrained

Check and specify the tip constraints is important. If the tip has set all constraints, then it follows the target in the x, y, z directions and keeps the same orientation. We have to pay attention to properly constrain the tip. But there exist some cases, when it is not possible. In this case we must use the dumped calculation method and we must specify the dumping factor. We need to note, if we check the Alpha-Beta-Gamma constraints in the dialog window, then the tip will try to take up the same orientation as the target has and if the Gamma constraint is unchecked, the matched tip z-axis and target z-axis orientation will freely rotate around the z-axis.

Chapter 5

Implementation

This chapter contains how we implement the individual parts of the design. We describe the process how we realized 3D model, how AutoCAD works with 2D and 3D objects, rotational inertia for every robot link, robot description format structure. We will discuss the problem of closed loop chain in URDF and show our robot 3D model. After that we describe Lilli in CoppeliaSim, robot control via remote API, physics engine setup and inverse kinematic solver. We show inverse kinematics calculations for the right arm and the solution to the dumping problem.

5.1 Building the clean model

5.1.1 AutoCAD and 2D parts

At the beginning, we had at our disposal DXF files from Lilli's author. This files contain 2D model of robot parts. Every single part we need to transform from 2D to 3D shape. By the help of AutoCAD we will easily extrude 2D objects to 3D. But at this point we had a big problem because every single part consisted of only lines and we needed to join this lines. After joining these parts we got closed spline but sometimes extrusion does not work because of the self-intersecting curve. This problem is solved by BOUNDARY command in AutoCAD which creates a region or polyline from an enclosed area. The other and a primitive solution to resolve self-intersecting curve is to find the line which causes the intersection. After all parts were extruded into 3D shape we built body parts by using every single object, translation and rotation.

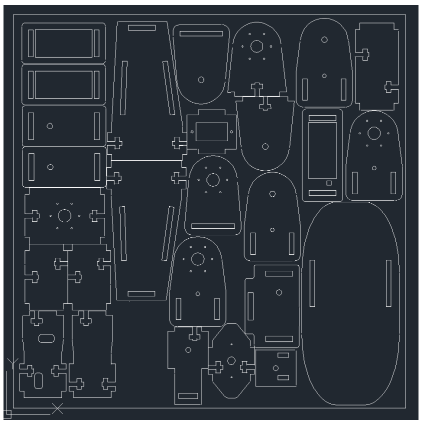


Figure 5.1: DXF files containing 2D objects

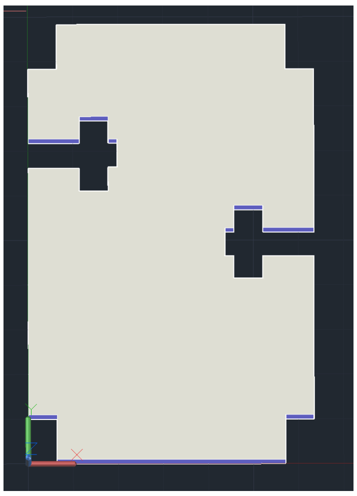


Figure 5.2: A single extruded part

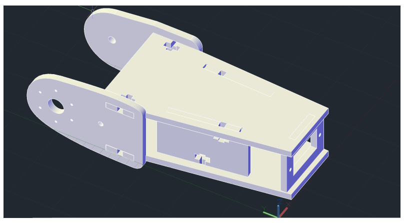


Figure 5.3: Right forearm

5.1.2 Problem of inertia and utilization of Solidworks

At this point we have all robot manipulators as STL files. We need to describe the model with URDF file format. First we defined the base link which is the pelvis of the robot. After that, from the base link we define the other parent-child relationships between the parts. As the links visual we used the exported STL files. The links origin is not defined yet. Because of that, every part x, y, z position are at the same origin. One of the solutions is we must calculate the right origin for each part by distances and by translation matrix we transform each part to the right origin. This calculation takes a lot of calculations and time, but this is not the only problem what we need to solve. The second problem is the inertial definition of the link element. This definition contains the x, y, z position of the center of mass, the mass itself and the 6 above-diagonal elements of rotational inertia matrix. If we want to calculate the inertia rotation matrix manually, we need to solve the integral

$$I = \int (x^2 + y^2)dm = \int (x^2 + y^2)pdV$$

for each modeled part. Of course we should be careful about the rotation axis. To solve this problem we used another CAD program which is called Solidworks. During work, we realized that Solidworks is more flexible than AutoCAD, because it solves all our problems mentioned above. It is true, that AutoCAD allows us to compute the center of mass and the moment of inertia for each link, but Solidworks is capable of a little more. After we have done the research about Solidworks abilities we came to the differences between them. Solidworks allows us to calculate the mass properties by the density part parameter and the visualization of CoM or the origins is much better than in AutoCAD. Another reason is that Solidworks solves the origin problem too. The last reason was the URDF exporter plugin. This plugin allows us to export the URDF file from Solidworks and it eases building the tree structure.

5.1.3 3D model

We knew we would continue to use Solidworks. We have each part exported not only as STL file but as DWG file. Solidworks allows us to import other CAD models. From each part we create a SLDPRT file where we set the material to Beech and this gives the part density 560 kg/m^3. When we have each part saved in PART representation Solidworks format, then we draw the servos and the horns. We use 3 types of servos:

- MG996R servo motor - the stall torque parameters are

  4.8V -> 9.4 [kgf·cm] -> 0.9218251 [N·m]
  6V -> 12 [kgf·cm] -> 1.176798 [N·m]

- MG90S servo motor - the stall torque parameters are

  4.8V -> 1.8 [kgf·cm] -> 0.1765 [N·m]
  6V -> 2.2 [kgf·cm] -> 0.215 [N·m]

- DS3225 digital servo motor - the stall torque parameters are

  4.8V -> 21 [kgf·cm] -> 2.05844 [N·m]
  6V -> 24.5 [kgf·cm] -> 2.4 [N·m]

We use the MG90S servo for the wrist and fingers joint because we need small force to move these joints. The DS3225 digital servo is at both knees and both ankles because the upper parts are heavier and the joint needs more force to load. Next we create assemblies for each parts which contain connections to the part, the right servo and the servo horn. An example of one assembly is shown at Figure 5.4.

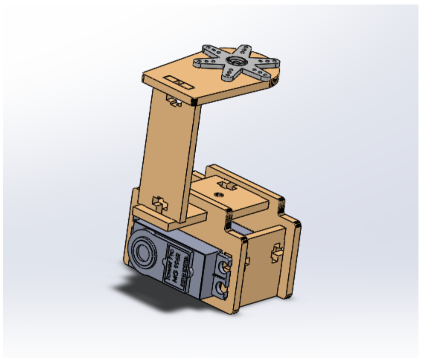


Figure 5.4: Robot left hip assembly

Next we build the 3D model from these assemblies. At first, we add the base link (pelvis) to the model. Solidworks uses mates to define the geometry relationship between assemblies. At the beginning, we mate the base link origin with the origin of the assembly. After that we add the other parts and define the relationship between them via mates.

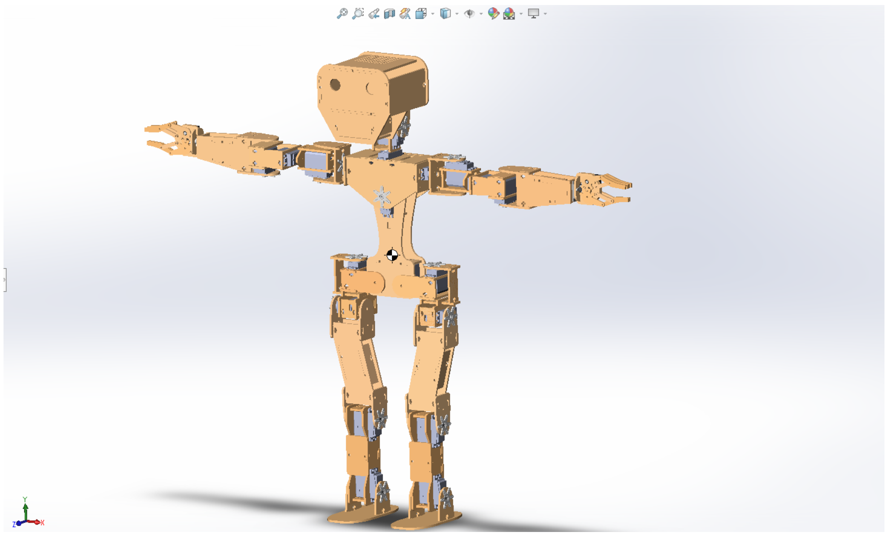


Figure 5.5: Lilli's 3D model

Building the 3D model was difficult because we had to understand the logic behind the model. After all we successfully built the 3D model which was shown in Figure 5.5. When the 3D model was completed, we defined the joint position. Lilli has 25 DOF and because that we define 25 coordinate systems for the joints. As you see in the next picture the joint coordinate system is visualized around the rotation axis.

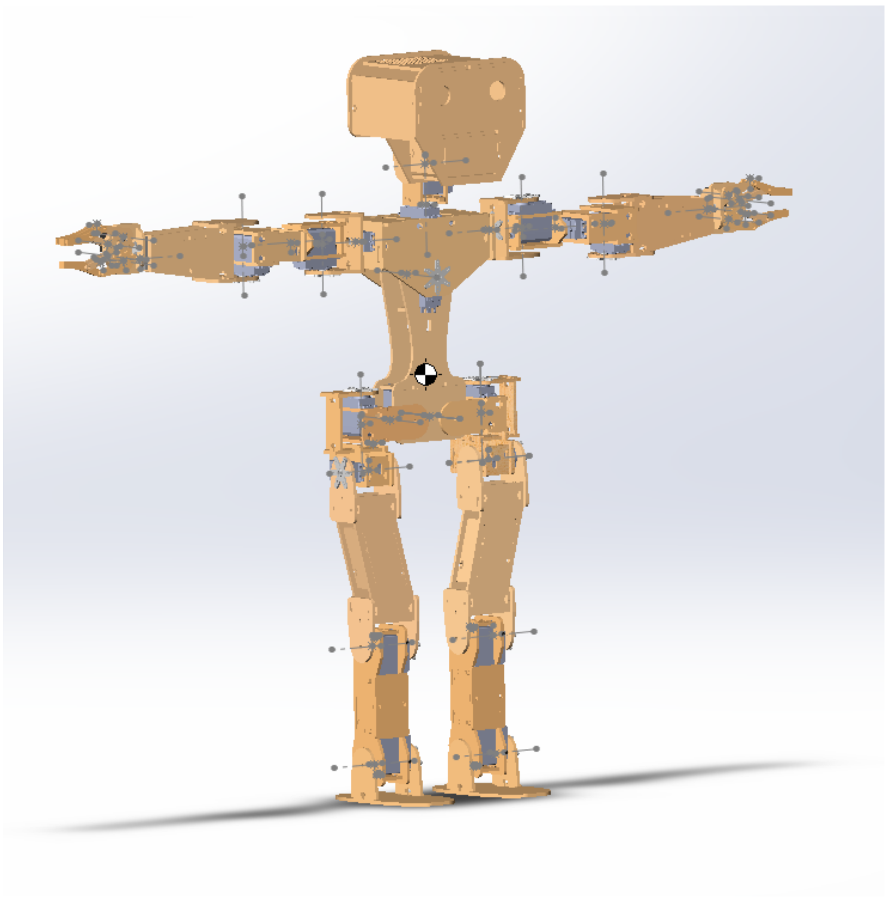
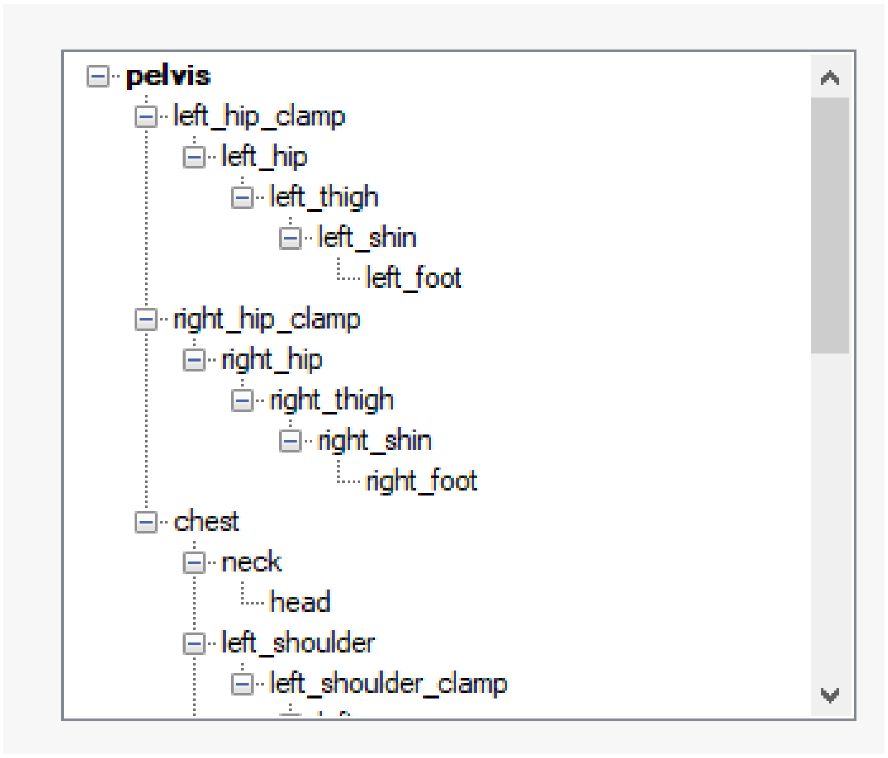


Figure 5.6: Visualised joint axis

Note, in the picture for the gripper there exists only one joint, but we must define them more to show the problem of closed loop chain. We are going to discuss this problem in the next section.

5.1.4 Lilli's description and closed loop chain problem

At this point we have the 3D model and the coordinate system generated for each joint. The next step was, that we applied the URDF exporter plugin and generated the URDF file format. Solidworks allows us to define the link name, the parent link, the joint name which connects the links, reference coordinate system for the link, reference axis for the joint and the joint type. After we defined the parent-child relationships we got the following tree structure



```
pelvis
    left_hip_clamp
        left_hip
            left_thigh
                left_shin
                    left_foot
    right_hip_clamp
        right_hip
            right_thigh
                right_shin
                    right_foot
    chest
        neck
            head
        left_shoulder
            left_shoulder_clamp
```

Figure 5.7: Lilli's tree structure

Next we exported the URDF file. URDF file contained all robot structure with the right joint origins. The visual mesh was usable too, we did not have to translate or rotate the origin directly in the description. Only the joints limit element was not completed and we have done it manually. The following listing shows all the aspects of the description format

```
<robot name="LilliHumanoid">
  <link name="pelvis">
    <inertial>
      <origin xyz="0.045652 -0.01564 0.078635" rpy="0 0 0" />
      <mass value="0.14677" />
      <inertia
        ixx="0.00021437"
        ixy="2.6116E-07"
        ixz="-1.2765E-07"
        iyy="0.00021725"
        iyz="-4.3704E-06"
        izz="4.9494E-05" />
    </inertial>
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://LillyAll/meshes/pelvis.STL" />
      </geometry>
      <material name="">
        <color rgba="0.75294 0.75294 0.75294 1" />
      </material>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://LillyAll/meshes/pelvis.STL" />
      </geometry>
    </collision>
```

```
</link>
<link name="left_hip_clamp">
  <inertial>
    <origin xyz="0.043299 -0.010259 0.0055623" rpy="0 0 0" />
    <mass value="0.048224" />
    <inertia
      ixx="2.0451E-05"
      ixy="4.3281E-08"
      ixz="3.9112E-07"
      iyy="1.5296E-05"
      iyz="1.4602E-07"
      izz="2.2881E-05" />
  </inertial>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh
          filename="package://LillyAll/meshes/left_hip_clamp.STL" />
    </geometry>
    <material name="">
      <color rgba="0.75294 0.75294 0.75294 1" />
    </material>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh
          filename="package://LillyAll/meshes/left_hip_clamp.STL" />
    </geometry>
    </collision>
  </link>
  <joint name="l_hip_z" type="revolute">
    <origin xyz="0.070848 -2.5108E-05 0.022508" rpy="0 0
        -0.00035439" />
    <parent link="pelvis" />
    <child link="left_hip_clamp" />
    <axis xyz="0 1 0" />
    <limit effort="1.17" lower="-0.523598775598"
        upper="0.497418836818" velocity="8.0"></limit>
  </joint>
</robot>
```

The above shown code snippet represents the base pelvis link and the left hip clamp link connected via revolute joint. Other links and joints are similar to this notation with own parameters. The joint limit element we define as follows. The effort was the stall torque parameter which we mentioned in previous section. The upper and lower parameter are the minimum and maximum angles in radians. The velocity parameter was calculated from the servo velocity parameter, where

- MG996R servo motor - the velocity parameter is

  4.8V -> 0.17 [sec/60°] -> 352.94 [degree/sec]
  -> 58.823 [RPM] -> 6.1599 [rad/s]
  6V -> 0.13 [sec/60°] -> 461.53 [degree/sec] ->
  76.9216 [RPM] -> 8.0552 [rad/s]

- MG90S servo motor - the velocity parameter is

4.8V -> 0.1 [sec/60°] -> 600 [degree/sec] ->
100 [RPM] -> 10.47 [rad/s]
6V -> 0.08 [sec/60°] -> 750 [degree/sec] ->
125 [RPM] -> 13.08 [rad/s]

- DS3225 digital servo motor - the velocity parameter is

4.8V -> 0.15 [sec/60°] -> 400 [degree/sec] ->
66.66 [RPM] -> 6.98 [rad/s]
6V -> 0.13 [sec/60°] -> 461.53 [degree/sec] ->
76.9216 [RPM] -> 8.0552 [rad/s]

A big problem of the URDF description is, that it does not allow closed loop chains. The closed loop chain problem is that two different joints have the same child link. In our case we have the gripper, which can behave this way.

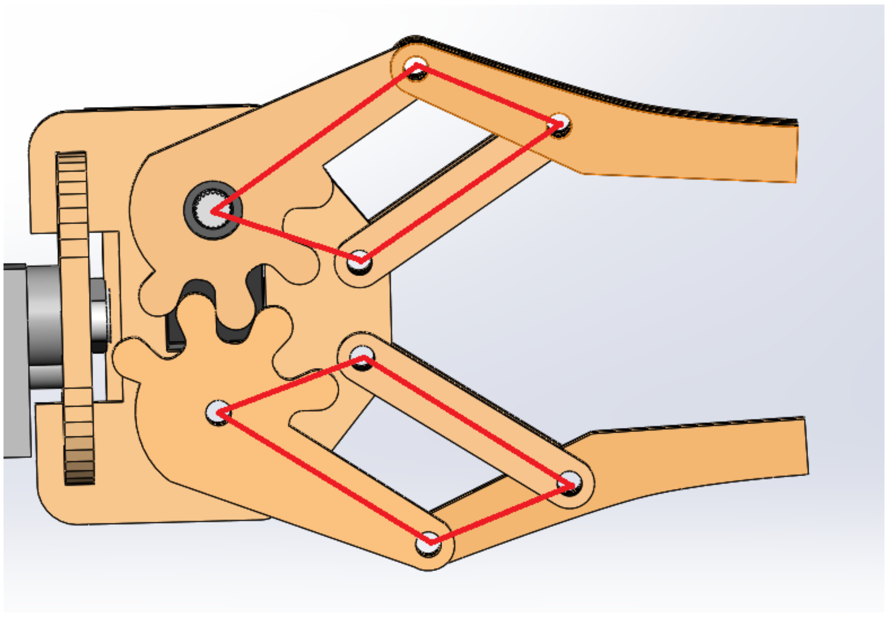
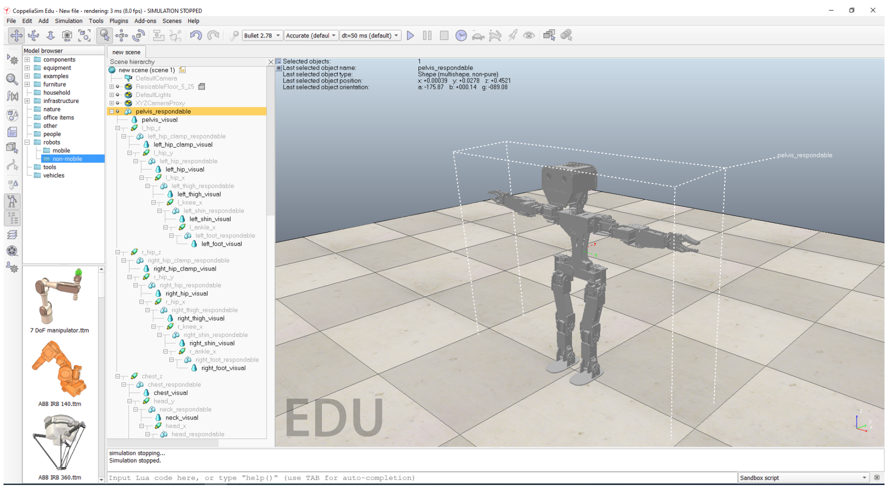


Figure 5.8: Closed loop chain

URDF does not support closed loop chain. For the future work we have 4 different ways to update it. The first choice is the pegasus gazebo plugins package which was developed for this purpose, but if we choose this option then we need to make the simulations in Gazebo instead of CoppeliaSim. The second option is convert all URDF format to SDF format, because SDF format allows the closed loop chains. The third option is to import the gripper directly in CoppeliaSim and after that to solve the closed loop. At the end, the easiest solution is to remove the inside links in the rhomboid and the outside links need to be fixed. If we remove the inside links then the model will not look like as the real robot and the gripper will not close according to the parallel links.

5.2 Lilli in CoppeliaSim

We have the 3D model and the URDF description and we import the URDF to the CoppeliaSim. See Figure 5.9.


CoppeliaSim Edu - New file - rendering: 3 ms (8.0 fps) - SIMULATION STOPPED
File Edit Add Simulation Tools Plugins Add-ons Scenes Help
Bullet 2.78
Accurate (defaul
dt=50 ms (default)
Model browser
new scene
Scene hierarchy
EDU
Sandbox script

```
Selected objects:                   1
Last selected object name:          pelvis_respondable
Last selected object type:          Shape (multishape, non-pure)
Last selected object position:      x: +0.00039   y: +0.0278   z: +0.4521
Last selected object orientation:   a: -175.87   b: +000.14   g: -089.08
```

```
simulation stopping...
Simulation stopped.
Input Lua code here, or type "help()" (use TAB for auto-completion)
```

Figure 5.9: Scene after the URDF import

On the right side there is the page which has one view and on the left side the model definition with the robot manipulators. Our shapes are respondable and dynamic and they have set the rotation, orientation and the moment of inertia from the URDF. Our joints default setting is torque/force mode which is needed to the simulation for the remote API control. When we start the simulation at this point then the robot falls on the floor, but we made sure that the right icons are showed next to the joints and shapes. That means than we build the clean model right and they are dynamically enabled. Preparing for the remote API control we must enable for each joint the control loop in the joint dialog window.

5.2.1 JAVA Remote API

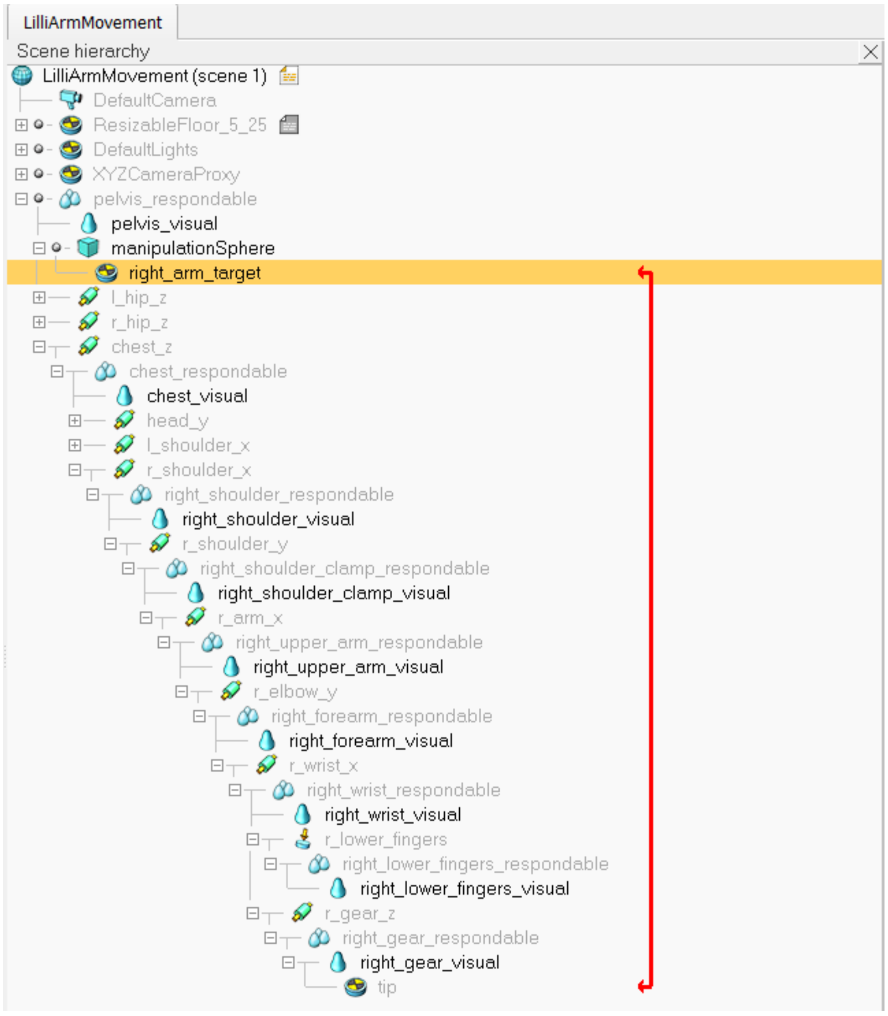
Remote API method allows the user to connect to CoppeliaSim with an external application. The remote API has two types: the b0-based remote API and the legacy remote API. The application support is various for these types, but we selected the legacy remote API and Java client to create the connection. This version has less dependencies but is more difficult to extend. We have 3 Java classes: Main.java, Initializer.java and LilliRemote.java. The Main is simple, this is called the Initializer which initializes the whole connection. It starts the server which returns a clientID that we are using during the simulation control. After we are connected to the server we get objects from the scene and handle the values. Our goal was to make movement via this API and we move the joint right arm with the following code

```
System.out.println("Start arm moving");
       String[] array = new String[]{"r_shoulder_y",
          "r_shoulder_x", "r_arm_z", "r_elbow_y"};
       int[] handles = new int[]{0,0,0,0};
       IntW objH = new IntW(1);

       for (int i = 0; i < array.length; i++) {
```

```
    vrep.simxGetObjectHandle(clientID, array[i], objH,
       vrep.simx_opmode_blocking);
    handles[i] = objH.getValue();
}

System.out.println("Joint handles are: " +
    Arrays.toString(handles));

float[] targetPositions = new float[]{(float)
    (90*Math.PI/180), (float) (85*Math.PI/180), (float)
    (90*Math.PI/180), (float) (45*Math.PI/180)};
float[] iniatialPositions = new float[]{(float)
    (0*Math.PI/180), (float) (0*Math.PI/180), (float)
    (0*Math.PI/180), (float) (0*Math.PI/180)};

for (int i = 0; i < handles.length; i++) {
    FloatW jointTargetPos = new FloatW(targetPositions[i]);
    vrep.simxSetJointTargetPosition(clientID, handles[i],
       jointTargetPos, vrep.simx_opmode_oneshot_wait);
}
```

5.2.2 Arm movement via inverse kinematics

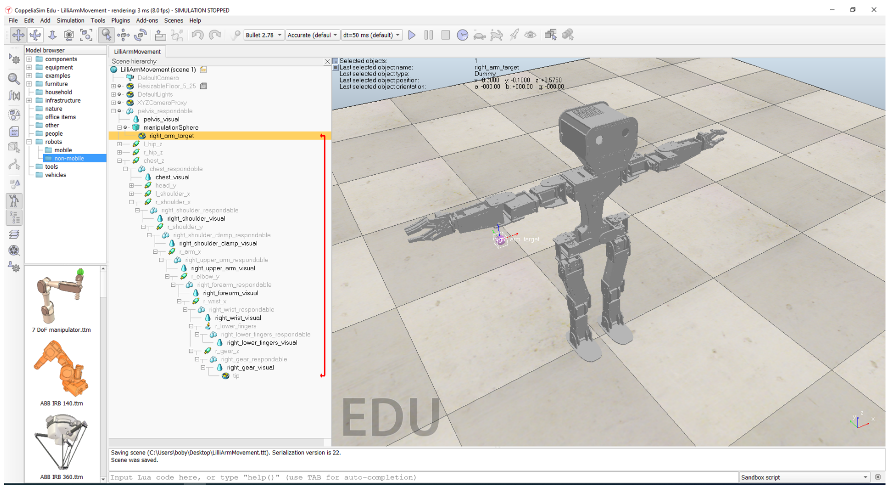
We would have liked to simulate how the right arm catches any object by inverse kinematics. For this purpose we needed to set some setting to robot model. At first, we allowed the inverse kinematics mode for the joints in the kinematic chain. Next we defined two dummies, the first is the end-effector which we added as a child to the last element in the kinematic chain which is the right gear visual object. The second dummy we named as right arm target, whose position and orientation are followed by the tooltip. After these steps we linked these dummies to IK, tip-target link.



```
LilliArmMovement
Scene hierarchy
LilliArmMovement (scene 1)
    DefaultCamera
    ResizableFloor_5_25
    DefaultLights
    XYZCameraProxy
    pelvis_respondable
        pelvis_visual
        manipulationSphere
            right_arm_target
        l_hip_z
        r_hip_z
        chest_z
            chest_respondable
                chest_visual
                head_y
                l_shoulder_x
                r_shoulder_x
                    right_shoulder_respondable
                        right_shoulder_visual
                        r_shoulder_y
                            right_shoulder_clamp_respondable
                                right_shoulder_clamp_visual
                                r_arm_x
                                    right_upper_arm_respondable
                                        right_upper_arm_visual
                                        r_elbow_y
                                            right_forearm_respondable
                                                right_forearm_visual
                                                r_wrist_x
                                                    right_wrist_respondable
                                                        right_wrist_visual
                                                        r_lower_fingers
                                                            right_lower_fingers_respondable
                                                                right_lower_fingers_visual
                                                        r_gear_z
                                                            right_gear_respondable
                                                                right_gear_visual
                                                                    tip
```

Figure 5.10: Linked tip-target pair

In the picture above, you can see the right linked dummies in the scene hierarchy. To improve the right arm manipulation we moved the `right_arm_target` and the tip to layer 11 to make dummies invisible. After that we created a sphere called *manipulationSphere*, which we assigned as the `right_arm_target` parent. The sphere was not dynamic and respondable. We moved the sphere at the same position as the target has and at the simulation we can drag the sphere instead of the target itself. We changed a few other details to the sphere for a nicer appearance. At this state we have the kinematic chain with an end-effector and the target and it looks like as


CoppeliaSim Edu - LilliArmMovement - rendering: 3 ms (8.0 fps) - SIMULATION STOPPED
File Edit Add Simulation Tools Plugins Add-ons Scenes Help
Bullet 2.78
Accurate (defaul
dt=50 ms (default)
Model browser
LilliArmMovement
Scene hierarchy
EDU

```
Selected objects:                   1
Last selected object name:          right_arm_target
Last selected object type:          Dummy
Last selected object position:      x: 0.3000  y: -0.1000  z: +0.5750
Last selected object orientation:   a: -000.00  b: +000.00  g: -000.00
```

```
Saving scene (C:\Users\boby\Desktop\LilliArmMovement.ttt). Serialization version is 22.
Scene was saved.
```

Figure 5.11: The scene with the sphere as the target object

When the scene was defined, then we set the IK groups and IK elements. In the calculation modules dialog, we created an IK group. The IK group has two types of calculation methods: Pseudo inverse and DLS (Damped least squares) calculation method. The first method uses the linear approximation of the inverse kinematics problem, which is the Jacobian Pseudo-inverse method. It is a widely used method in the field of robotics, but often has instability near singularities. The second method is more stable and it is finding the value of $\Delta\Theta$ that minimises the quantity. Except the calculation method we were able to manipulate with other parameters as maximum iteration, calculation weights etc. We selected the Pseudo inverse method and set the maximum iterations to 6. Next we added an IK element for the group, which is the tip. For the tip we checked all the constraints to follow the target position and orientation. In the last step, we ran the simulation and we translated the sphere by the mouse to see the inverse kinematic solution for the joints.

5.2.2.1 Solution to unstable model

With the settings above, the calculations will be fast, but the above Pseudo inverse method becomes unstable when the target is out of reach or the kinematic chain is overconstrained etc. In this case we can use the second method which is the DLS. When we selected the DLS method and after it ran the simulation, then the model was more stable. To reach the most stable model we would find the best damping factor and the maximum iterations count. This solution has a big disadvantage, then the DLS method is much slower than the Pseudo inverse. There exists the best solution for this problem. We defined two IK groups. One of the groups calculates the results with Pseudo inverse method and the second IK group calculates the results with the DLS method. Both IK groups have the same IK element. For the IK groups with the DLS method we gave condition to perform the method only when the Pseudo inverse method fails. It is true, then we must found the best damping factor and iterations for the DLS method. After that we ran the simulation again and we tested whether the model is still unstable. When the sphere was too far from the tooltip, the model stayed stable.

Chapter 6

Evaluation

In this chapter we evaluate the results of the diploma thesis. We start with the 3D model, after that we show the URDF. At the end, you see warm-ups of the model with the robot's arm and leg.

6.1 3D model

The aim of this section will be create our real robot 3D model, which is used in the simulator. The robot has 25 degree of freedom and has 26 links and 25 joints. The model was built by CAD systems AutoCAD and Solidworks. The relations between the links and joints were defined by mates. For all joint was defined a rotation axis according to which the joint rotates. We successfully created the 3D model, which resembles to the real one. On the Figure 6.1 you seen the 3D model which are used in simulations.

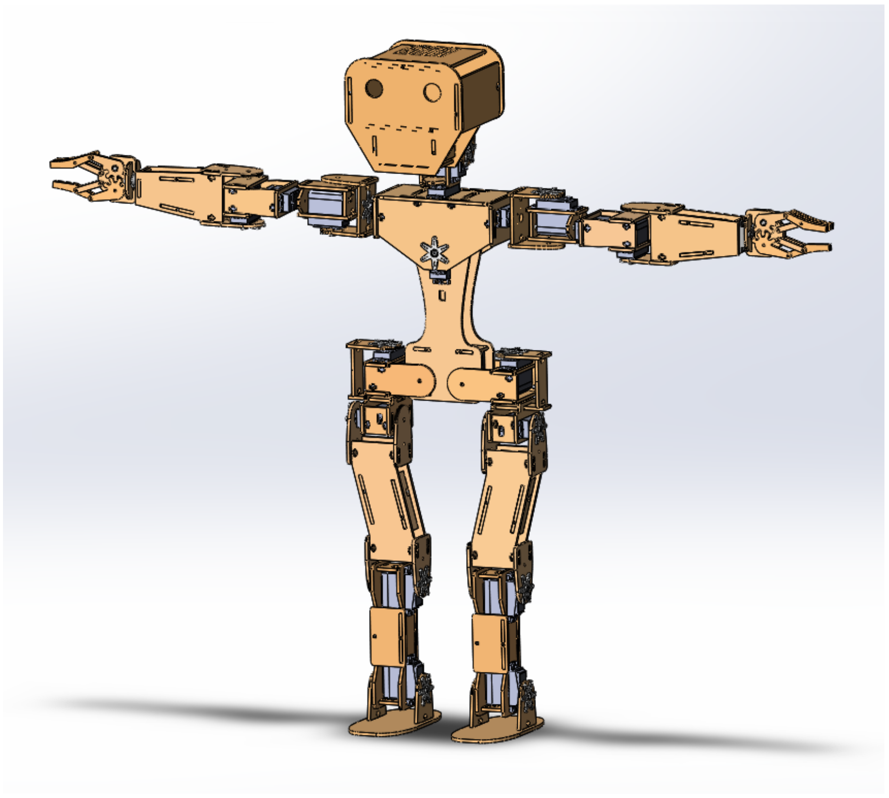


Figure 6.1: 3D model

6.2 URDF

The aim of this section will be to create a unified description format of the robot. The selected description format was the URDF, which uses XML format. The URDF contains the model definition and allows to import the model for every simulators which support URDF files. The URDF describes the links mass properties like rotational inertia, center of mass and mass. Next it includes the visual mesh and the geometry mesh. On the other hand, servomotors are described by the joint element and its properties like child, parent, origin, axis, effort and velocity of the joint. In the below shown code we see the left thigh and the left shin links and the knee revolute joint connection.

```
<link
    name="left_thigh">
    <inertial>
      <origin
        xyz="-0.0044252 -0.088607 0.00011125"
        rpy="0 0 0" />
      <mass
        value="0.051871" />
      <inertia
        ixx="0.00013801"
        ixy="-2.0243E-05"
        ixz="3.1231E-09"
        iyy="3.3032E-05"
        iyz="3.7526E-08"
        izz="0.00015531" />
    </inertial>
    <visual>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://LillyHumanoid/meshes/left_thigh.STL" />
      </geometry>
      <material
        name="">
        <color
          rgba="0.75294 0.75294 0.75294 1" />
      </material>
    </visual>
    <collision>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://LillyHumanoid/meshes/left_thigh.STL" />
      </geometry>
    </collision>
  </link>
<link
    name="left_shin">
    <inertial>
      <origin
        xyz="-0.015221 -0.059782 -0.00033796"
        rpy="0 0 0" />
      <mass
        value="0.1032" />
      <inertia
        ixx="8.8877E-05"
        ixy="1.4139E-07"
        ixz="5.9253E-09"
        iyy="2.4534E-05"
        iyz="-1.0795E-06"
        izz="9.4679E-05" />
    </inertial>
    <visual>
```

```
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://LillyHumanoid/meshes/left_shin.STL" />
      </geometry>
      <material
        name="">
        <color
          rgba="0.75294 0.75294 0.75294 1" />
      </material>
    </visual>
    <collision>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://LillyHumanoid/meshes/left_shin.STL" />
      </geometry>
    </collision>
  </link>
  <joint
    name="l_knee_x"
    type="revolute">
    <origin
      xyz="0 -0.17729 9.8422E-05"
      rpy="-1.5467E-05 0.0057687 0" />
    <parent
      link="left_thigh" />
    <child
      link="left_shin" />
    <axis
      xyz="1 0 0" />
    <limit
      lower="-0.0610865238198"
      upper="2.33874119767"
      effort="2.4"
      velocity="8" />
  </joint>
```

6.3 Warmp-up with leg

The aim of this section will be to test the robot model with a simple movement. This example shows the leg movement. We used the remote API aspect of the robot controlling. We laid down the model on the floor. Then we initialized the connection between the Java client and the CoppeliaSim simulator. With the simxGetObjectHandle remote API function we saved all the joint handles that we needed. Next we defined the target positions of the joints and after that we called the simxSetJointTargetPosition which moved the joint to the position, which we sent as a parameter. Because the joint is revolute we had to send the angle in radians. The below shown code represents the warm-up of the leg.

```
System.out.println("Start left leg moving");
String[] array = new String[]{"l_hip_y", "l_knee_y", "l_ankle_y"};
```

```
int[] handles = new int[]{0,0,0};
IntW objHandle = new IntW(1);

for (int i = 0; i < array.length; i++) {
    vrep.simxGetObjectHandle(clientID, array[i], objHandle,
        vrep.simx_opmode_blocking);
    handles[i] = objHandle.getValue();
}

System.out.println("Joint handles are: " +
    Arrays.toString(handles));

float[] targetPositions = new float[]{(float) (-75*Math.PI/180),
    (float) (-45*Math.PI/180), (float) (30*Math.PI/180)};

for (int i = 0; i < handles.length; i++) {
    System.out.println("move");
    FloatW jointTargetPos = new FloatW(targetPositions[i]);
    vrep.simxSetJointTargetPosition(clientID, handles[i],
        jointTargetPos, vrep.simx_opmode_oneshot_wait);
}

vrep.simxFinish(clientID);
```

6.4 Warmp-up with arm

The aim of this section will be to test the robot model arm movement. In this example we used the remote API and Java client too. First we initialized the connection. After it, we saved the joint handles for the joints. Next we defined the targetPositions array to save the target values in radians. Before the simulation we had to check that the joint is in torque/force mode, the motor is enabled and the control loop is enabled too. If the joint will be moved too fast while the simulation then you can set the upper velocity limit. At the end, we call the simxSetJointTargetPosition to move the joints. The below shown code represents the warmp-up of the leg.

```
System.out.println("Start arm moving");
String[] array = new String[]{"r_shoulder_y", "r_shoulder_x",
    "r_arm_z", "r_elbow_y"};
int[] handles = new int[]{0,0,0,0};
IntW objH = new IntW(1);

for (int i = 0; i < array.length; i++) {
    vrep.simxGetObjectHandle(clientID, array[i], objH,
        vrep.simx_opmode_blocking);
    handles[i] = objH.getValue();
}

System.out.println("Joint handles are: " +
    Arrays.toString(handles));

float[] targetPositions = new float[]{(float) (90*Math.PI/180),
    (float) (85*Math.PI/180), (float) (90*Math.PI/180), (float)
    (45*Math.PI/180)};
float[] iniatialPositions = new float[]{(float) (0*Math.PI/180),
    (float) (0*Math.PI/180), (float) (0*Math.PI/180), (float)
    (0*Math.PI/180)};
```

```
for (int i = 0; i < handles.length; i++) {
    FloatW jointTargetPos = new FloatW(targetPositions[i]);
initialPosition.getValue());
    vrep.simxSetJointTargetPosition(clientID, handles[i],
        jointTargetPos, vrep.simx_opmode_oneshot_wait);
}

sleep(5000);

for (int j = 0; j < handles.length; j++) {
    FloatW jointInitialPosition = new FloatW(iniatialPositions[j]);
initialPosition.getValue());
    vrep.simxSetJointTargetPosition(clientID, handles[j],
        jointInitialPosition, vrep.simx_opmode_oneshot_wait);
}

vrep.simxFinish(clientID);
```

Chapter 7

Conclusion

The aims of the diploma thesis were creation of the 3D model for simulation purpose and testing of the algorithms in simulations and after that on the real robot. Next we need to explore and implement algorithms by which the robot will be able to move in its environment, including inverse kinematics and the use of machine learning algorithms.

We started with a wide research of humanoid robots as well as a history overview, how to model the shapes, the joint types, degree of freedom of the robot, the mass properties and the forward and inverse kinematics problems. In the first part of the proposal chapter we discussed how to build the clean model for the simulation purpose and which unified structure to use for it. This section was successfully implemented and this was described in the implementation. In the second part of a proposal we described the CoppeliaSim simulator scene, model definition, simulation control via embedded scripts or remote API and after that we explained how inverse kinematics solver works with kinematic chains. We have successfully created an example of a simple robot arm movement via remote API and Java client. Moreover, we created one kinematic chain defining the right arm and we simulated an object catching with the arm.

Implementation and algorithm for the robot movement is difficult. In the future, we can extend our capabilities of robot motion path creation and test it on a real robot. Important thing will be to solve the closed loop chain problem which we mentioned at the implementation chapter.

Bibliography

[AL09] Andreas Aristidou and Joan Lasenby. Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver, 09 2009.

[BBM$^+$02] Rodney Brooks, Cynthia Breazeal, Matthew Marjanovic, Brian Scassellati, and Matthew M. Williamson. The cog project: Building a humanoid robot. *Lecture Notes in Artificial Intelligence*, 1562, 03 2002.

[Beh08] Sven Behnke. Humanoid robots - from fiction to reality? *KI*, 22:5–9, 01 2008.

[BM18] L.V. Bharath and Himanth M. Forward kinematics analysis of robot manipulator using different screw operators. *International Journal of Robotics and Automation*, 3, 03 2018.

[Bor16] Havrila Boris. Simulácia humanoidného robota. 2016.

[Copa] CoppeliaSim. Building a clean model tutorial.

[Copb] CoppeliaSim. Joint types.

[dLGCZM04] Javier de Lope, Rafaela González-Careaga, Telmo Zarraonandia, and Darío Maravall. Inverse kinematics for humanoid robots using artificial neural networks. volume 2809, pages 448–459, 04 2004.

[Don] Bruce Randall Donald. Forward kinematics: The denavit-hartenberg convention.

[Edw] Lin Edwards. Armar-iii, the robot that learns via touch.

[KB06] Serdar Kucuk and Z. Bingul. *Robot Kinematics: Forward and Inverse Kinematics*. 12 2006.

[PGPW18] Lenka Pitonakova, Manuel Giuliani, Anthony Pipe, and Alan Winfield. Feature and performance comparison of the v-rep, gazebo and argos robot simulators. 02 2018.

[RPFS13] Robert B. Leighton Richard P. Feynman and Matthew Sands. *The Feynman Lectures on Physics*, volume I. 1963-1965, 2006, 2013.

[SG] Robert Haschke Stuart Glaser, William Woodall. Xacro file format.

[SRR11] Nima Shafii, Luís Reis, and Rosaldo Rossetti. Two humanoid simulators: Comparison and synthesis. pages 1 – 6, 07 2011.

[SZK17] Roman Szewczyk, Cezary Zielinski, and Malgorzata Kaliczynska. *Automation 2017: Innovations in Automation, Robotics and Measurement Techniques*, volume 550. 01 2017.

[TPS17] Dr.Sweet Chandan Tarun Pratap Singh, Dr.P.Suresh. Forward and inverse kinematic analysis of robotic manipulators. volume 04, pages 1459–1469, 02 2017.

[TS00] G. Tevatia and S. Schaal. Inverse kinematics for humanoid robots. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 1, pages 294–299 vol.1, April 2000.

Attachment

All files are available on github: `https://github.com/Robotics-DAI-FMFI-UK/cu-lIllI`. The folder name is `lilli_LTS`.

SD card - AutoCAD files, SOLIDWORKS PART and ASSEMBLY files, 3D model, URDF, STL meshes, CoppeliaSim .ttm file, java files for remote API client, README