SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Ciudad Guzmán

INSTITUTO TECNOLÓGICO DE CD. GUZMÁN

TITULACIÓN INTEGRAL
TESIS

TEMA:

MODEL AND CONTROL OF A HUMANOID ROBOT ON ROS ENVIRONMENT

QUE PARA OBTENER EL TÍTULO DE:
INGENIERO ELECTRÓNICO

PRESENTA:
MIRIAM FLORES PADILLA

ASESOR(A):
DR. JESÚS EZEQUIEL MOLINAR SOLIS

CD. GUZMÁN JALISCO, MÉXICO, FEBRERO DE 2019

Cd. Guzmán, Municipio de Zapotlán el Grande, Jal, 12/Febrero/2019

ASUNTO: Liberación de Proyecto para Titulación Integral.

M.C. FAVIO REY LUA MADRIGAL
JEFE DE LA DIVISION DE ESTUDIOS PROFESIONALES
P R E S E N T E

Por este medio le informo que ha sido liberado el siguiente proyecto para la Titulación Integral:

Nombre del Egresado:	MIRIAM FLORES PADILLA
Carrera:	INGENIERIA ELECTRONICA
No. De Control:	13290397
Nombre del Proyecto:	MODEL AND CONTROL OF HUMANOID ROBOT ON ROS ENVIRONMENT
Producto:	TITULACIÓN INTEGRAL (Tesis)

Agradezco de antemano su valioso apoyo en esta importante actividad para la formación profesional de nuestros egresados.

A T E N T A M E N T E

[illegible]

M.E.H. MARCO ANTONIO SOSA LÓPEZ
JEFE DEL DEPTO. ELÉCTRICA Y ELECTRÓNICA

[illegible]	[illegible]	[illegible]
DR. JESUS EZEQUIEL MOLINAR SOLIS ASESOR	DR. RAMÓN CHAVEZ BRACAMONTES REVISOR	DR. HUMBERTO BRACAMONTES DEL TORO REVISOR

ESTADOS UNIDOS MEXICANOS
S.E.P. TecNM
INSTITUTO TECNOLOGICO
DE CD. GUZMAN
DEPTO. ELECTRICA Y
ELECTRONICA

C.p.expediente
DLAS/MASL/adc

Abstract

This document explains the advantages of the creation of a humanoid robot model inside the Gazebo simulator, with the help of the ROS environment and how to load a simple sample control routine. This project pretend to explain the load of models in a free software dedicated to robots development. This software allows to test the performance of real robots saving money.
The first chapter will give an introduction to the problem that will lead the investigation, explains the reasons of how this project will be elaborated, the limitations and expectations.
In chapter number two, it is briefly explained the background of the robotics, the goals already achieved in the present days and the scopes that robotics are expected to have in the future days. It is important to notice that the most part of the goals achieved by the robots nowadays is done by private companies, and the reason of this project, as explained in the first chapter, is to boost the free development of this knowledge.
The chapters three and four talk about how the project was done and the results of the investigation; the proceedings and its steps are explained in the chapter three under the name "Methodology" as some of the results that are needed to explain the following parts of the processes, the chapter four "Results" explain the rest of the result and some test that can be made to review the correct performance of the model.

Acknowledgements

First of all, I want to say how important to me was to had the help and backup of my family, mom, dad, sister, and brother in law, I want you to know that without you this project had never been able to begin. Thank you to my dance family, Eduardo and Elizabeth, for being there for me in the hardest moments, to remind me that "I still want it all", and even more important, that I am able to achieve it. My career friends, thank you for letting me learn too much of you, I'm a better person and will be able to be an even more complete professional with all those conversations and advises. To my friends of Robotekp, thank you for helping me all those times and all those laughs in the right moments.
Thank you!

Contents

List of Figures

Listings

Chapter 1

Overview

1.1 An approach to the problem

Robotics is a discipline little present in the Instituto Tecnológico de Ciudad Guzmán curricula, even though it could be a professional and managerial departure with so much presence in the future. The robot word has its origins in a classic book of Karel Capek, RUR (Robots Universal Rossum). It makes reference to those helpers as robota, a word in Czech that means forced job or slave. The robots will be a product in mass when they can join the society in the same environment we humans live. For example, it would be so much easier to displace a robot with wheels, but then it will not reach scenes with stairs or obstacles. To help a people to get up will be better to use a crane robot, but it will not be able to cook or set up the table. The format which adapts the better to our environment is humanoid, given that it has been designed by and for the human.[UOC, 2012]
For this reasons, the need to learn and develop humanoid robotics, with managerial and educational aims is a natural conclusion. Furthermore, we realize the need of different ways of interacting with robotics, without expensive investments. Usually, the public educational centers do not have the resources for purchase the necessary materials, so to can practice control routines in free software, for the teachers and the students, would be very useful.

1.2 Justification

This document pretends to explain the development of the modeling the humanoid robot Mex-One in the way of the environment ROS and simple routines of control inside the Gazebo simulator, to be able to prove, without actually risking the real robot, routines of control and see the reaction to different physic properties such as gravity, friction, and damping.
The project is going to be developed using the Gazebo simulator, on which, all robot pieces and their properties like mass are going to be loaded with the help of CAD tools through the ROS environment. Then, the controllers are going to be loaded in the robot ROS package and, going to be tested by sending different commands to the robot joint.
All this will be made to achieve a simulator in which some the students of the Automatic Control faculty of the CINVESVAV, Gdl. can prove different theories of control, created by them.

1.3 Objective

1.3.1 General Objective

Modeling of the humanoid robot Mex-One in the Gazebo simulator with ROS environment and simple example controls created in ROS and visualized in Gazebo.

1.3.2 Specific Objective

- Modeling of the humanoid robot Mex-One in the Gazebo simulator.
- Create a controller for every joint of the robot.
- Can move and control every joint of the robot.
- Based in a cpp file, be able to send simple example control routines.

1.4 Scope and limitations

1.4.1 Scope

For the modeling, a prototype of the Mex-One humanoid robot is going to be created inside the Gazebo simulator in ROS environment, it will be able to emulate the physical reactions of the real robot, thanks to all the specifications of the robot physics (characterization of links and joints). Different simple examples of control routines are going to be tested inside the simulator, to review the correct operation of the simulated robot, and the accurate load of all the physical properties.

1.4.2 Limitations

This project will be limited by the following causes:

- Scattered information about how to load models in Gazebo.
- Lack of time to develop complex control routines. Need of advisors in the Gazebo subject.
- The limited economic resources to stay in the investigation center for a longer period of time.

1.5 Hypothesis

It is possible to create a model of a humanoid robot inside a simulator, capable of emulating the physical qualities of the real robot, developed on free software, for control theories tests.

Chapter 2

Reference framework

2.1 Historical framework

Robotics is the branch of Engineering that deals with the application of computer science to the design and use of machines with the objective that the result can somehow substitute the people in the realization of determined functions or tasks.
Those machines are often used nowadays in the commercial and industrial field to effectuate exact tasks, and of course because it implies a cheaper workforce than the human. Even it is used to carry out the most disgusting tasks that humans refuse to do because they are very heavy, dangerous or unbearable. In the industrial plant, it is common to see a robot to displace and to carry out tasks such as assembly, packing, and transfers, among others.
From the beginning a discipline and as a fundamental part of the engineering, robotics have been tirelessly searching to construct artifacts that materialize the wish of the humans to create beings to their likeness, to delegate to them the heavy and disgusting tasks, works and activities. But even though no many expect it, since immemorial, far far away from the computer, there were a few expressions of robotics. For example, ancient Egyptians joint mechanical arms to the statue of some of their gods and wielding that the movement was made by work and grace of them, even the greek constructed statues that operated with hydraulics systems, which were used to fascinate the church lovers.[Ucha, 2009]
If human-like machines are the first images of robots both in the earlier con-

cepts, automata and in Asimov's science fiction, in reality, humanoids are the last kind of robots that came into existence. The pioneers in humanoid robots are the WABOT (Figure 2.1) developed by Ichiro Kato [Kato, 1986].

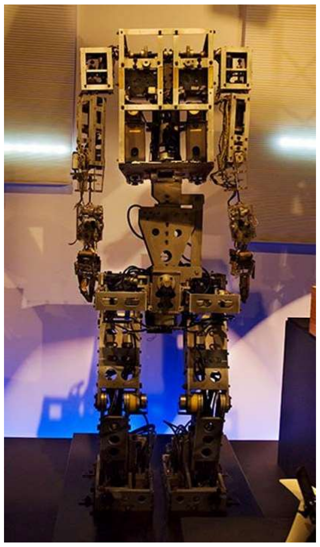


Figure 2.1: WABOT-1 (1973) First humanoid robot, developed in the Waseda University.

WABOT-1 was essentially built from 2 arms mounted atop of a WL-5 (Waseda Leg 5) which was a continuous effort from the same group since 1967. WABOT-1 inherited all the advances in locomotion of WL-5 including balancing and walking. The first walking robot was still at the primary development with only quasi-static stepping which results in a 45-second long footstep. The robot also featured vision sensor, tactile sensors and a speech synthesizing system and was reported to have "the mental faculty of a one-and-haft-year-old childâĂİ. [Dang, 2012]
Nowadays the technological advancements can be demonstrated by enumerating the abilities of some of the most amazing humanoid robots abilities and it is different capabilities, such as Sophia, developed by Handson Robotics, the first humanoid robot in getting a citizenship (in the kingdom of Saudi Arabia) because of its ability of logical conversations and gestures according to the conversation, making her look almost human [Retto, 2017]; Atlas developed by Boston Dynamics, that has great independent movement ca-

pabilities and even can make pirouettes in the air without losing stability when falling [Dynamics, 2018]; ASIMO by Honda Robotics, able of agile autonomous mobility and interaction with their environment, making decisions based on the same [Honda, 2018]; DRC HUBO developed by University of Nevada, Las Vegas, that has the ability of driving a vehicle, handle different short of tools, climb stairs and even moving around winding terrain [of Engineering, 2018].

2.2 Conceptual framework

2.2.1 Ubuntu

Ubuntu is a distribution of GNU/Linux, an operating system focused in personal computers (desktop and laptops), it is one of the most important distributions of Linux globally. The distribution's name came to the concepts Zulu an Xhosa of Ubuntu, that means "humanity towards other" and "I am because we are".
Examples of operative systems are Microsoft Windows XP, MS Windows Vista, MS Windows 98, OS X (MAC), MS-Dos, Unix and Linux in this case. Inside Linux are some distributions, the most popular in the occidental world are Ubuntu, RedHat, Mandriva, and SuSe.
Ubuntu includes a series of many programs: Pidgin for instant messaging (MSN, Yahoo, Gtalk, etc.); firefox web navigator; for the creation and edition of documents brings OpenOffice, programs to manage photography (cameras), to record music, listen to music and watch videos, etc., apart from a large list of free software that allow you to do almost any task.[Martínez, 2008]

Ubuntu 16.04

Ubuntu 16.04 is built on the 4.4 series of Linux Kernels, released in January of 2016. Includes a native kernel module for ZFS, an advanced filesystem originating in the 2000s at Sun Microsystems and currently developed for Open Source systems under the umbrella of the OpenZFS project. ZFS combines the traditional roles of a filesystem and volume manager, and offers many compelling features.
The Apt tools have not changed a great deal, although Ubuntu 16.04 upgrades to Apt 1.2, which includes some security improvements. Users migrating from older releases may also wish to consider use of the ***apt*** com-

mand in place of the traditional ***apt-get*** and ***apt-cache*** for many package management operations. More detail on the ***apt*** command can be found in Package Management Basics: apt, yum, dnf, pkg.
Ubuntu 16.04 includes a native kernel module for ZFS, an advanced filesystem originating in the 2000s at Sun Microsystems and currently developed for Open Source systems under the umbrella of the OpenZFS project. ZFS combines the traditional roles of a filesystem and volume manager, and offers many compelling features. Comes by default with Python 3.5.1 installed as the ***python3*** binary. [Bearnes, 2016]

2.2.2 ROS

As the full name of Robot Operating System suggests, ROS is an operating system for robots. In the same way as operating systems for PCs, servers or standalone devices, ROS is a full operating system for service robotics.
It provides not only standard operating system services (hardware abstraction, contention management, process management), but also high-level functionalities (asynchronous and synchronous calls, centralised database, a robot configuration system, etc.).
The main idea of a robotics OS is to avoid continuously reinventing the wheel, and to offer standardised functionalities performing hardware abstraction, just like a conventional OS for PCs, hence the analogous name.
ROS is developed and maintained by a Californian company, Willow Garage, formed in 2006 by Scott Hassan. Their idea is that if we want to see robots reach our homes, then research needs to be accelerated by providing solid hardware and software bases that are open source.
ROS's philosophy can be summarised in the following five main principles:

- **Peer to Peer:**
  A peer-to-peer architecture coupled to a buffering system and a lookup system (a name service called "master" in ROS), enables each component to dialogue directly with any other, synchronously or asynchronously as required.

- **Multi-language:**
  ROS is language-neutral, and can be programmed in various languages.Peer-to-peer connections are negotiated in XML-RPC, which exists in a great

number of languages. To support a new language, either C++ classes are re-wrapped or classes are written enabling messages to be generated.

- **Tools-based:**
  Rather than a monolithic runtime environment, ROS adopted a microkernel design, which uses a large number of small tools to build and run the various ROS components. The advantage of this system is that a problem with one executable does not affect the others, which makes the system more robust and flexible than a system based on a centralised runtime environment.

- **Thin:**
  To combat the development of algorithms that are entangled to a lesser or greater degree with the robotics OS and are therefore hard to reuse subsequently, ROS developers intend for drivers and other algorithms to be contained in standalone executables. This ensures maximum reusability and, above all, keeps its size down.

- **Free and open source:**
  We have already explained the reasons for this choice. ROS passes data between modules using inter-process communications and, as a result, modules do not need to be linked within a single process, thereby making the use of different licences a possibility.

The list of ROS-compatible robots grows constantly. However, it is worth mentioning the best-known, namely NAO, Lego Mindstorms NXT, IRobot Roomba, TurtleBot and last but definitely not least, Willow GarageâĂŹs iconic PR2.
ROS resources are organised into a hierarchical structure on disc. Two important concepts stand out:

- **The package:**
  The fundamental unit within ROS software organisation. A package is a directory containing nodes (nodes are explained below), external

libraries, data, configuration files and one xml configuration file called manifest.xml.

- **The stack:**
  A collection of packages. It offers a set of functionalities such as navigation, positioning, etc. A stack is a directory containing package directories plus a configuration file called stack.xml.

Mention should nevertheless be made of (another) interesting contribution to robotics from ROS in the shape of URDF (Unified Robot Description Format), an XML format used to describe an entire robot in the form of a standardised file. Robots described in this way can be static or dynamic and the physical and collision properties can be added to it.
Besides the standard, ROS offers several tools used to generate, parse or check this format. URDF is used by the Gazebo simulator, for example, to represent the robot.[Mazzari, 2016]

2.2.3 Gazebo

Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs.
Typical uses of Gazebo include:

- Testing robotics algorithms.
- Designing robots.
- Performing regression testing with realistic scenarios.

A few key features of Gazebo include:

- Multiple physics engines.
- A rich library of robot models and environments.
- A wide variety of sensors.
- Convenient programmatic and graphical interfaces.[Foundation, 2014]

2.2.4 URDF

URDF is an abbreviation of Unified Robot Description Format and is a format of a robot model described in XML notation. In the robot model considered in URDF, joints are defined as ***Joints***, and rigid bodies joined by joints are defined as ***Links***.
Benefits of URDF:

- Open link mechanism can be described.
- It is possible to describe the relative position relation of links.
- Ability to define geometric elements and motion elements of independent robot.
- The description is comparatively easy.
- Convenient tool called Xacro (macro description).

Disadvantages of URDF:

- Description of closed link mechanism is impossible (not supported).
- It is impossible to describe the absolute position relation of the link.
- It is impossible to define geometric elements and motion elements of multiple robots.
- Impossible to define environment such as lighting and altitude map. [Tanaka, 2016]

2.2.5 SDF

SDF Models can range from simple shapes to complex robots. It refers to the <*model*> SDF tag, and is essentially a collection of links, joints, collision objects, visuals, and plugins. Generating a model file can be difficult depending on the complexity of the desired model. This page will offer some tips on how to build your models.

- ***Link:***
  A link contains the physical properties of one body of the model. This can be a wheel, or a link in a joint chain. Each link may contain many collision and visual elements. Try to reduce the number of links in your models in order to improve performance and stability.

  - ***Collision:***
    A collision element encapsulates a geometry that is used to collision checking. This can be a simple shape (which is preferred), or a triangle mesh (which consumes greater resources). A link may contain many collision elements.
  - ***Visual:***
    A visual element is used to visualize parts of a link. A link may contain 0 or more visual elements.
  - ***Inertial:***
    The inertial element describes the dynamic properties of the link, such as mass and rotational inertia matrix.

- ***Joints:***
  A joint connects two links. A parent and child relationship is established along with other parameters such as axis of rotation, and joint limits.

- ***Plugins:***
  A joint connects two links. A parent and child relationship is established along with other parameters such as axis of rotation, and joint limits.

Chapter 3

Methodology

The project was planned to be done with ROS Indigo, Ubuntu 14.04 and Gazebo 2.0, but during its realization and the investigation, was found that a newest version of the software would be more suitable, due the information able to this kind of work was reference, mainly, to ROS Kinetic, Ubuntu 16.04 and Gazebo 7.0.

3.1 Get used to basic Ubuntu commands.

Some of the basic functions used in the realization of the investigation and its functions are described below:

- cd / direction: To move among the directories.
- man [page]: Help of a specific command.
- ls: Details about the current folder.
- cp [dir1/file] [dir2]: Copy the file in the first direction (dir1), to the second direction (dir2).
- mv [dir1/file] [dir2]: Move the file in the first direction (dir1), to the second direction (dir2).
- touch [name]: Create a new file under the name given in space "name".
- echo: Print text in the terminal.

- pwd: Print in the terminal the directory of the file under work.
- file: Print the format of a file.
- gedit: Open a window to edit a file.

3.2 Grow an SDF with Gazebo.

To grow an SDF file with Gazebo, it is assumed that you have knowledge of how the pieces (or links, as will be referred to them in the present document) of the model are assembled, its degrees of freedom and axis of rotation in all the joints. It is also assumed, that you have possession of all the files of the 3D figures needed to assemble the model.
Gazebo has two known ways to charge models in the simulator (without external software help), the first one, is to write the code of an SDF, which involves a good knowledge of the positions, rotations, and center of each one of the links in a Cartesian plane, and it is made without a visual feedback during the elaboration of the code. The knowledge necessary to successfully create a model in the first option is easier to have if you create the 3D figures of all the links and assembled then in a software made to dressing those figures, but as the present investigation was made without this knowledge, the second option was the most suitable. In the second one is to change the model in a visual way with help of a Gazebo tool called "Model Editor".
The "Model Editor" is a Gazebo tool able only in the version 6.0 and following, it is used to assemble different links in a visual framework, and allows the user to create links of basic figures, such as cylinders, spheres, and cubes; allows the user to create links of 3D figures saved previously in the user computer. Those figures must be saved in Stl, Obj, or Dae format to can be used in Gazebo. The Dae and Obj formats allows the user to give textures to the links, while the Stl format only allows the user to give color to the links. In the present document, due to practicality, the Stl format is preferred, because of the lightness of the format.

3.2.1 Import Links.

To start working in the model editor, you must open a Gazebo 6 or an upper version, in the elaboration of the present document, the 7.0 version was chosen because of its compatibility with ROS. In the upper tools bar, will be found the "Edit" option, and inside this option, the "Model Editor" is placed, as the (Figure 3.1) shows.

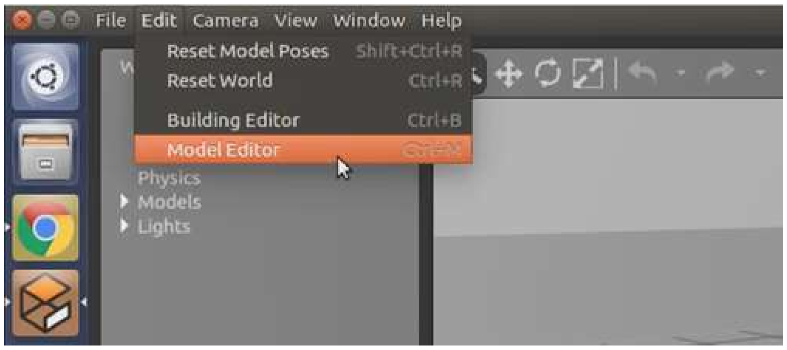
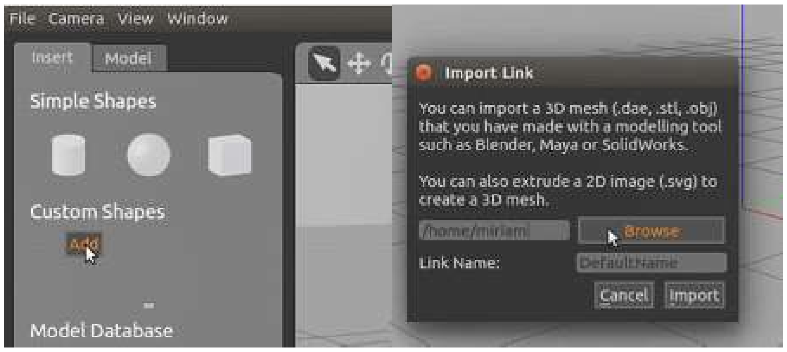


Figure 3.1: Model Editor menu Gazebo 7.0.

This option will open the "Model Editor" tool that will allow importing Stl files as Links to Gazebo, this is achieved by directing to the left panel where the option "Customer Shapes" and its button "Add" stands, the "Add" button opens a window under the name "Import Link" that allows the user to browse to a file, that will be used as a Link, and a space to change the Link name, but unfortunately, in the version 7.0 of Gazebo this option does not work correctly, since no matter the name given in this space, the Link name will be set by the default procedure of Gazebo. The access to the "Custom Shapes" and the window unfold by its "Add" button are shown in the (Figure 3.2).



You can import a 3D mesh (.dae, .stl, .obj) that you have made with a modelling tool such as Blender, Maya or SolidWorks.

You can also extrude a 2D image (.svg) to create a 3D mesh.

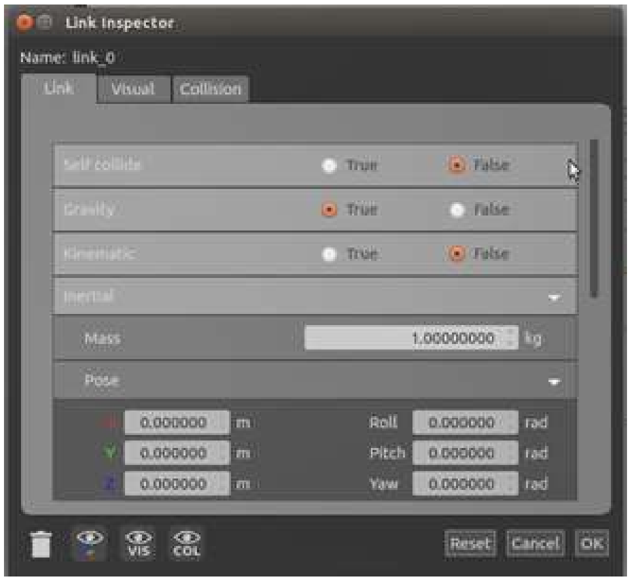
/home/miriam

Link Name: DefaultName

Figure 3.2: Custom Shapes and Import Link window.

3.2.2 Modify Link parameters.

To modify the physic parameters of each Link, the "Model Editor" has another tool, called "Link Inspector" which enables the user to rewrite the parameters of each Link with the real parameters. Those parameters can be seen in the development software of the 3D figures as their physical properties, such as the center of mass, inertia, and weight. That window is opened by right click the Link and choose the option "Open Link inspector" on the unfolded submenu. The window that will pop out, shown in (Figure 3.3), on its "Link" flange have default values for the fields to modify according to the information of each Link.



Name: link_0

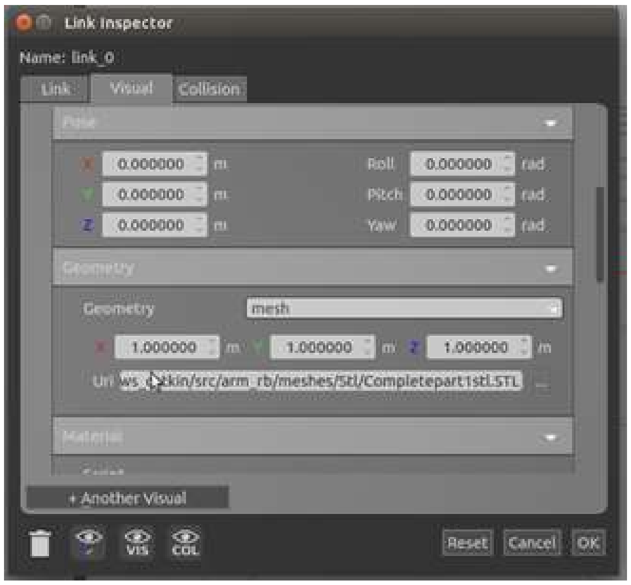
Self collide: True [✓ False]
Gravity: [✓ True] False
Kinematic: True [✓ False]
Mass: 1.00000000 kg

X	0.000000	m	Roll	0.000000	rad
Y	0.000000	m	Pitch	0.000000	rad
Z	0.000000	m	Yaw	0.000000	rad

Figure 3.3: Link flange of the Link Inspector menu.

It also allows the user to chose to activate or deactivate physic characteristics for each link that will affect the general performance of the model, cause of that, it is recommended to put the "Self collide" option on true, this will allow the model to know when one of its Links is colliding with another; "Gravity" on true, to allow the mass of the Link be affected by the gravity; and "Kinematic" on false, to allow the Link to have movement.
In the "Visual" flange the options to set a color, a visual origin of the object, and a general position of that origin, are not the only important parameters to modify, the formats allowed by Gazebo do not have the format of its uni-

ties, because of that, all the units are converted to meters, what can cause an incorrect charge of the proportions of the model. In such cases, the "Visual" flange has options to change the Link's dimensions, as is shown in the menus of the (Figure 3.4), on the x, y, and z axles; in the present document, a re-dimension of 0,001 in all the Link axles was made, because they were made with millimeter scale in units, an as was said before, Gazebo take each unit as meter due the kind of format used in Gazebo, doesn't allow information about the type of units used.



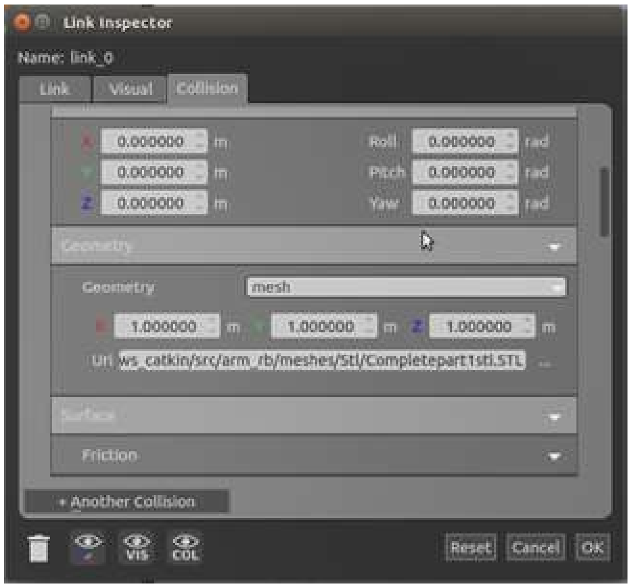
Name: link_0

X 0.000000 m Roll 0.000000 rad
Y 0.000000 m Pitch 0.000000 rad
Z 0.000000 m Yaw 0.000000 rad

Geometry mesh
X 1.000000 m Y 1.000000 m Z 1.000000 m
Uri ws_c[illegible]kin/src/arm_rb/meshes/Stl/Completepart1stl.STL

Figure 3.4: Visual flange of the Link Inspector menu.

The "Collision" flange, as the visual, allows the user to set an origin of the Stl used to review the collisions, re-dimension the Link, and also, use a different 3D figure for the collision review, due a well detailed figure will make the simulation slower than a not that well-shaped figure, cause of the number of faces in the object, the longest the number of faces in the collision figure, the slower the relation between the real-time and the simulation-time will be. This new object can be load by the "Uri" space in the "Geometry" options as the (Figure 3.5) shows, where you can browse any other object in Stl, Dae or Obj format, as when the Link was charged.
It also allows the user to set some physic properties of the material of the real robot pieces, like the friction coefficients $\mu1$ y $\mu2$, contact stiffness (Kp) and damping (Kd) for rigid body contacts, among others.



Name: link_0

X 0.000000 m Roll 0.000000 rad
Y 0.000000 m Pitch 0.000000 rad
Z 0.000000 m Yaw 0.000000 rad

Geometry mesh

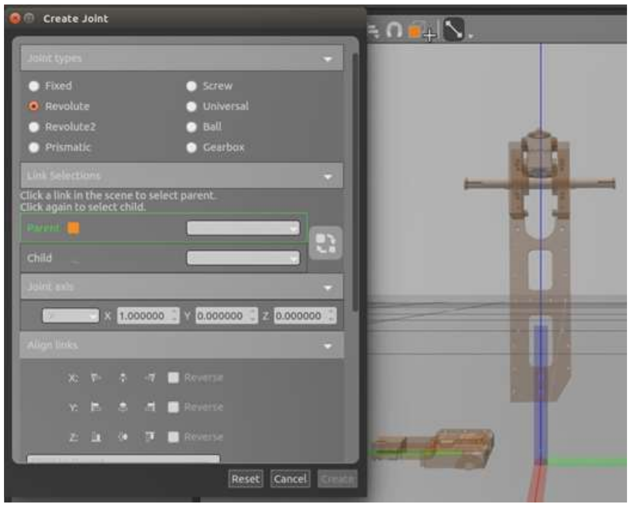
X 1.000000 m Y 1.000000 m Z 1.000000 m

Uri ws_catkin/src/arm_rb/meshes/Stl/Completepart1stl.STL

Figure 3.5: Collision flange of the Link Inspector menu.

3.2.3 Create Joints

Each couple of links require a "Joint" to assemble them, that is to say, if a model has 24 Links, has to have 23 Joints. Such join can be "Fixed" for stick the links to each other, "Revolute" to let one spin in one axis of the other, "Revolute2" to let one to spin in two axles of the other, "Prismatic" to allow one to displace in an axis of the other, "Screw" to make a joint with one freedom degree and coupled displacement, "Universal" that allows the movement between each other like a couple of magnet balls, "Ball" to move one as a ball in its socket, being the first link the socket, and "Gearbox" which allows setting the velocity relation between the links.
A joint can be created by going to the upper toolbar, to the last option "Create a Joint" which will pop out a window with the same name, as the (Figure 3.6) shows, that will allow the user to chose between the different types of joints; the link that will be the "Parent" and the "Child" which is to say who can move with the other as a reference; the axis or axles of reference to the movement; in necessary case, the alignments to line up the child link in reference with the parent link; the pose (in cartesian coordinates) of the joint regarding the parent link; and the relative pose of the child link regarding the parent link.



Joint types: [✓ Revolute]

Joint axis: X 1.000000 Y 0.000000 Z 0.000000

Figure 3.6: "Create a Joint" window.

To make a correct joint it is necessary that both coordinates axis are united in the same place on the space, and well-centered so that the joint can rotate correctly, as is shown in the (Figure 3.7), otherwise the joints will seem to be rotating with a point out of the model as a reference. In the case of a "Revolute" joint, the axis of rotation will appear to have a yellow arrow in it rotation axis, in the (Figure 3.7), the axis of rotation is the y-axis.

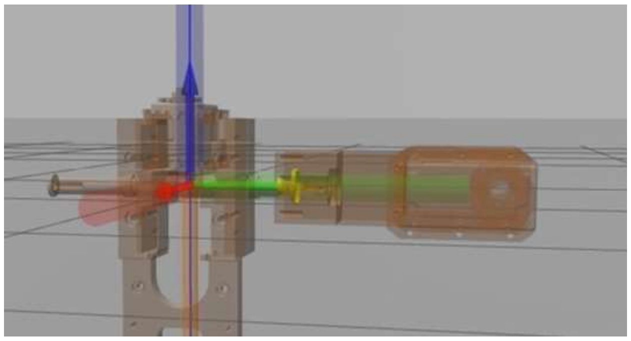


Figure 3.7: Well-centered joint with rotation in the y-axis.

3.2.4 SDF location.

A finished model can be stored by closing the "Model Editor" in the "File" menu, the "Exit model editor" option, and giving the model the chosen name at saving. This will open the model in the simulation window of Gazebo, and create the SDF file in the folder called "model_editor_models" in a subfolder called with the name you to given to the model, under the name "model.sdf". This folder will also have a file called "model.config", that file allows Gazebo to charge the model in the simulator, but to charge the model with ROS, the file is unimportant.

3.3 Build a URDF.

To build a URDF without the previously mentioned knowledge of the positions, rotations, and center of each one of the links in a Cartesian plane, and with the help of the SDF file of the model, this section will explain how to transform the information of that file into a URDF file, that ROS can call to Gazebo.

3.3.1 Start the URDF.

First of all, a new file with the extension .urdf is needed, in this case, a file called "model.urdf" was created. The first code line in the file has to state the file language, all the URDF files are created in XML; the second line must declare the beginning of the robot statement and its name; in the present case, a xacro statement was made at the same time to future work of it. The before said statements are declared as follows in Listing 3.1:

Listing 3.1: First statements.

```
<?xml version="1.0"?>
<robot name="mex_rb" xmlns:xacro="http://www.ros.org/wiki/xacro">
```

As it can be easily highly regarded, the name used for the robot in the present document was "mex_rb". It is important to highlight that the name of the robot can not have capital letters, or it will be seen as an error in the ROS launch stage.

3.3.2 Links statement.

To transform the information of the SDF to the URDF format, it is necessary to change some values and lost some qualities. The URDF format does not allow a general position for a link, because of this, in the first link, you have to change the values of the center of mass to be an addition of the general pose and the pose of the center of mass in the SDF, to use as the new value of the URDF, the second link can keep the values of the SDF without any adds, because of the repair that should be made in the "Joint" statements; the mass value and the inertia values can stay the same, but its format change. The Listing 3.2 shows the inertial statements of the SDF and the Listing 3.3 shows the URDF inertial statements of the same link.
The URDF format also losses the option to activate or deactivate the "Self Collide" , "Kinematic", and "Gravity" functions, and in both cases, a name must be declared for each link. All the links in the robot must have different names, and as is easy to observe, some of the ways of declaring the same things, change; it is the case of the position, named in the SDF as *pose frame* which includes in the same space the coordinates (x,y,z) and angles (r,p,y), and named in the URDF as *origin* with different spaces for the coordinates (x,y,z) and angles (r,p,y).

Listing 3.2: Inertial statements in SDF.

```
<link name='link_0'>
  <pose frame=''>-0.003958 0 1.00846 0 -0 0</pose>
  <inertial>
    <mass>0.598008</mass>
    <inertia>
      <ixx>0.005</ixx>
      <ixy>0.0</ixy>
      <ixz>0.0</ixz>
      <iyy>0.0045</iyy>
      <iyz>0.0</iyz>
      <izz>0.007</izz>
    </inertia>
    <pose frame=''>0.003958 0 -0.00846 0 -0 0</pose>
  </inertial>
  <self_collide>1</self_collide>
  <kinematic>0</kinematic>
```

Listing 3.3: Inertial statements in URDF.

```
<link name="link_0">
  <inertial>
    <origin xyz="0 0 1.00" rpy="0 -1.5707 -1.577"/>
    <mass value="0.598008"/>
    <inertia
      ixx="0.005"
      ixy="0.0"
      ixz="0.0"
      iyy="0.0045"
      iyz="0.0"
      izz="0.007"/>
  </inertial>
```

In the "Visual" statements, as in the "Inertial", the URDF format does not allow a general position for a link, and in the first link, you have to change the values of the coordinates to be an addition of the general pose and the pose of the visual coordinates in the SDF, to use the add as the new value of the URDF, the second link can keep the values of the SDF without any adds, as in the inertial values, because of the repair that should be made in the "Joint" statements.
The color of the link gets lost in the URDF, a color could be set, but it will not be read as a Gazebo color, and therefore, do not have any effect in the model launched by ROS to Gazebo, that is why the color statement is not used. The Listing 3.4 shows the visual statements of the SDF, and the Listing 3.5 shows the way the same information is declared in the URDF, taking into account that this is the first link of the model.

Listing 3.4: Visual statements in SDF.

```
  <visual name='visual'>
    <pose frame=''>-0.2531 -0.0781 -0.2355 1.5707 0 1.5707</pose>
    <geometry>
      <mesh>
        <uri>/home/miriam/.Stl/Completepart1stl.STL</uri>
        <scale>0.001 0.001 0.001</scale>
      </mesh>
    </geometry>
    <material>
      <lighting>1</lighting>
    <script>
      <uri>file://media/materials/scripts/gazebo.material</uri>
      <name>Gazebo/White</name>
    </script>
    <ambient>1 1 1 1</ambient>
    <diffuse>1 1 1 1</diffuse>
    <specular>1 1 1 1</specular>
    <emissive>0 0 0 1</emissive>
    <shader type='vertex'>
      <normal_map>__default__</normal_map>
    </shader>
  </material>
  <transparency>0</transparency>
  <cast_shadows>1</cast_shadows>
</visual>
```

Listing 3.5: Visual statements in URDF.

```
<visual>
  <origin xyz="-0.257058 -0.0781 0.77296" rpy="1.5707 0 1.5707"/>
  <geometry>
  <mesh
      filename="package://mex_rb/meshes/Stl/Completepart1stl.STL"
      scale="0.001 0.001 0.001"/>
  </geometry>
</visual>
```

The "Collision" statements, as in the statements before, in the first link, you have to change the values of the coordinates to be an addition of the general pose and the pose of the visual coordinates in the SDF, to use the add as the new value of the URDF, and the second link can keep the values of the SDF without any adds. This case is the most obvious about the loss of information because it almost loses all its statements, leaving it only with the geometry information, but in the following procedure will be explained how to recover some of this information to the model. The Listing 3.6 shows the collision statements of the SDF, and the Listing 3.7 shows the URDF way to declare the statements.

Listing 3.6: Collision statements in SDF.

```
<collision name='collision'>
    <laser_retro>0</laser_retro>
    <max_contacts>10</max_contacts>
    <pose frame=''>-0.2531 -0.0781 -0.2355 1.5707 0 1.5707</pose>
    <geometry>
      <mesh>
        <uri>/home/miriam/.Stl/CCompletepart1stl.STL</uri>
        <scale>0.001 0.001 0.001</scale>
      </mesh>
    </geometry>
    <surface>
      <friction>
        <ode>
          <mu>0.61</mu>
          <mu2>0.47</mu2>
          <fdir1>0 0 0</fdir1>
          <slip1>0</slip1>
          <slip2>0</slip2>
        </ode>
        <torsional>
          <coefficient>1</coefficient>
          <patch_radius>0</patch_radius>
          <surface_radius>0</surface_radius>
          <use_patch_radius>1</use_patch_radius>
          <ode>
            <slip>0</slip>
          </ode>
        </torsional>
      </friction>
      <bounce>
        <restitution_coefficient>0</restitution_coefficient>
        <threshold>1e+06</threshold>
      </bounce>
      <contact>
        <collide_without_contact>0</collide_without_contact>
        <collide_without_contact_bitmask>1</collide_without_contact_bitmask>
        <collide_bitmask>1</collide_bitmask>
        <ode>
          <soft_cfm>0</soft_cfm>
          <soft_erp>0.2</soft_erp>
          <kp>1e+09</kp>
          <kd>1</kd>
          <max_vel>0.01</max_vel>
          <min_depth>0</min_depth>
        </ode>
        <bullet>
          <split_impulse>1</split_impulse>
          <split_impulse_penetration_threshold>-0.01</split_impulse_penetration_threshold>
          <soft_cfm>0</soft_cfm>
          <soft_erp>0.2</soft_erp>
          <kp>1e+09</kp>
          <kd>1</kd>
        </bullet>
      </contact>
    </surface>
  </collision>
</link>
```

Listing 3.7: Collision statements in URDF.

```
  <collision>
    <origin xyz="-0.257058 -0.0781 0.77296" rpy="1.5707 0 1.5707"/>
    <geometry>
    <mesh
        filename="package://mex_rb/meshes/Stl/CCompletepart1stl.STL"
        scale="0.001 0.001 0.001"/>
    </geometry>
  </collision>
</link>
```

3.3.3 Joints statement.

The joints statement, as the other statements, loss some of the information, the most remarkable change in the joints, is that the origin of the joint, instead of stay in zero, is changed by the general position of the child link, in the first step of the family tree. On the second and bellow steps, the parent general position (current position) to the children general position (desired position) is subtracted, and the result is used as the origin of the joint. In this case, the example is not taken of the first joint, because it is "Fixed" and it will not show how to charge the axis values. The Listing 3.8 shows a joint statement in an SDF, the Listing 3.9 shows the same joint in URDF format.

Listing 3.8: Joint statements in SDF.

```
<joint name='link_0_JOINT_2' type='revolute'>
  <parent>link_0</parent>
  <child>link_3</child>
  <pose frame=''>0 0 0 0 -0 0</pose>
  <axis>
    <xyz>0 0 1</xyz>
    <use_parent_model_frame>0</use_parent_model_frame>
    <limit>
      <lower>-3.1416</lower>
      <upper>3.1416</upper>
      <effort>-1</effort>
      <velocity>-1</velocity>
    </limit>
    <dynamics>
      <spring_reference>0</spring_reference>
      <spring_stiffness>0</spring_stiffness>
      <damping>0.04</damping>
      <friction>0.4</friction>
    </dynamics>
  </axis>
  <physics>
    <ode>
      <limit>
        <cfm>0</cfm>
        <erp>0.2</erp>
      </limit>
      <suspension>
        <cfm>0</cfm>
        <erp>0.2</erp>
      </suspension>
    </ode>
  </physics>
</joint>
```

Listing 3.9: Joint statements in URDF.

```
 <joint name="link_0_JOINT_2" type="revolute">
  <parent link="link_0"/>
  <child link="link_3"/>
  <origin xyz="0.015042 0 1.05396" rpy="0 -0 0"/>
  <axis xyz="0 0 1"/>
  <limit effort="-1.0" velocity="1.0" lower="-3.1416" upper="3.1416"
     />
</joint>
```

3.3.4 Gazebo references.

To recover some of the lost information, you can write a code of Gazebo references for both links and joints. You can almost recover all the lost information, but the present document will just show how to recover some of the most important information. As Listing 3.10 shows, the information that the code is trying to recover are the friction coefficients $\mu 1$ y $\mu 2$, contact stiffness (Kp) and damping (Kd) for rigid body contacts, and the link color. On the other hand, the Listing 3.11 shows that the joints only recover the dynamic coefficients of friction and damping, the example does not take the first joint, because, as in the joint translation, the information is not useful for fixed joints.
Both references need to state at the beginning of its code the name of the subject of references and follow the structure of the fields of SDF to properly recover the information, otherwise, the information will not be taken as a statement for Gazebo.

Listing 3.10: Gazebo link references.

```
<gazebo reference="link_0">
  <selfCollide>true</selfCollide>
  <mu1>0.61</mu1>
  <mu2>0.47</mu2>
  <kp>10000000.0</kp>
  <kd>1.0</kd>
  <material>Gazebo/White</material>
</gazebo>
```

Listing 3.11: Gazebo joint references.

```
<gazebo reference="link_0_JOINT_2">
  <axis>
    <dynamics>
      <damping>0.04</damping>
      <friction>0.4</friction>
    </dynamics>
  </axis>
</gazebo>
```

3.4 Launch the URDF in Gazebo with ROS.

The present section will describe how to correctly use the file created in the section before, because of that, you should start creating a WorkSpace to develop ROS projects.

3.4.1 Create a WorkSpace in ROS.

To create a WorkSpace in ROS, you should execute on terminal the commands in the Listing 3.12, the first one calls the setup of the ROS environment; the second creates a folder called "mex_one" in the user's folder, and a subfolder called "src"; the third direct you to the first created folder; and the fourth run the ROS environment in the folder, which should create a series of folder that will help ROS to run all the projects that we will create in this folder.

Listing 3.12: Commands to create a WorkSpace.

```
  $ source /opt/ros/kinetic/setup.bash
  $ mkdir -p ~/mex_one/src
  $ cd ~/mex_one/
  $ catkin_make
```

Before creating a Workspace, it is necessary to create a package, that will contain our project. To accomplish this, you must introduce the code in the Listing 3.13 on the terminal, the first line direct you to the "mex_one" folder; the second set the ROS environment in the folder; the third direct you to the "src" folder, in which all the packages are created; the fourth create a package called "mex_rb" which will be dependent on the packages "gazebo_ros", "message_generation", "roscpp", "rospy", "std_msgs", "tf", "controller_manager", and "joint_state_controller", the fifth direct you to the "mex_one" folder; the sixth and last, compile the code created by the new package and makes it recognizable by ROS.

Listing 3.13: Commands to create a package.

```
$ cd ~/mex_one/
$ source devel/setup.bash
$ cd src/
$ catkin_create_pkg mex_rb gazebo_ros message_generation roscpp
   rospy std_msgs tf controller_manager joint_state_controller
$ cd ~/mex_one/
$ catkin_make
```

In the resultant folders of the above procedure, you must save the URDF model, to be accurate, in the package folder named "mex_rb" in a new subfolder called "urdf".

3.4.2 Create a Gazebo world.

The next step to be able to launch a model in Gazebo from ROS is to create a world in which our model exist, for it is necessary to create a file with extension .world, in this case, the file will be named "model.world", which will be placed in the folder "urdf", as the "model.urdf" file. The Listing 3.14 shows the code that must be written in the world file. The main function of the code is to create a default world of Gazebo, with a ground plane and a "sun", that is the source of light and reference for the cast shadows.

Listing 3.14: Code to create a Gazebo world.

```
<?xml version="1.0" ?>
<sdf version='1.6'>
  <world name="default">
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://sun</uri>
    </include>

  </world>
</sdf>
```

3.4.3 Create a launch file.

The launch file will allow you to see the model working in Gazebo, the Listing 3.15 shows the commands that must be written in a new file called "model.launch", placed in the "mex_rb" folder. The main function of the code is to create an empty Gazebo world, find and charge the arrangements made in the "model.world" file, start Gazebo in the debug mode, enable the user's interface, start with the simulation in pause, make the ROS "nodes" to utilize the Gazebo simulation time, make the "State log" record the model information, enable the Gazebo feedback, load the URDF into the ROS "Parameter Server ", and run a python script to the send a service call to "gazebo_ros" to spawn the URDF robot of model, launching it to Gazebo.

Listing 3.15: Code to create a launcher.

```
<?xml version="1.0"?>

<launch>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find mex_rb)/urdf/model.world"/>
    <arg name="debug" value="false" />
    <arg name="gui" value="true" />
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="verbose" value="true" />
  </include>

  <param name="robot_description"
   command="$(find xacro)/xacro --inorder '$(find
       mex_rb)/urdf/model.urdf.xacro'" />

  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
```

```
      respawn="false" output="screen"
    args="-file $(find mex_rb)/urdf/model.urdf.xacro -urdf -x 0 -y 0
       -z 0 -model model"/>

</launch>
```

The file named "model.urdf" must change its name to "model.urdf.xacro"

3.5 Add the controllers to de SDF.

To indicate to ROS and the model in Gazebo which joints have a motor, you must create "transmissions" for each joint with a motor, the code for one transmission that works under position commands is shown in the Listing 3.16. This code has to be included in the file "model.urdf.xacro" under the Gazebo references. The transmission code gives a new name that must be different for each transmission; it is stablished the simple interface transmission, which means that the joint is spin in one axis; the reference joint name; the type of interface that will be used to control the joint, in this case, position controllers; the name of the motor, that will change to the real name of the motor given by the manufacturer of the motor and its data acquisition card; the torque constant of the motor, also given by the manufacturer; and the mechanical reduction, in case of have a gearbox in the joint.

Listing 3.16: Code to create a transmission.

```
<transmission name="trans_1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="link_0_JOINT_2">
    <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="kinect_motor0">
    <motorTorqueConstant>1.0</motorTorqueConstant>
    <mechanicalReduction>1.0</mechanicalReduction>
  </actuator>
</transmission>
```

To complete the load of the controllers for the robot, it is necessary to create a Yaml file that will call controllers for each joint and set a PID controller with its gains easy to modify from this file. A file called "configuration.yaml" must be created in the "mex_rb" folder with the code of the Listing 3.17 written in it. The code set the namespace "model" to refer to the robot, load a joint state controller, and create a controller for the first joint called "kinect_controller0", which has the "position" type of the controller, to use in this case, the joint to whom it refers, and the PID gains. A "kinect_controller" must be created for each joint with a motor.

Listing 3.17: Code to create a Yaml file.

```
model:

  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  kinect_controller0:
    type: position_controllers/JointPositionController
    joint: link_0_JOINT_0_0
    pid:
      p: 2.0
      i: 0.5
      d: 0.1
```

3.6 Create the control files.

The controller file can be created in Phyton or in C++, in the present investigation, the C++ language was preferred, due to is the most used for the final users of the product of the investigation. You must create a file with the extension ".cpp " in the subfolder "src" of the folder "mex_rb". The following code in the Listing 3.18 is an example of a control that will make the first articulation to move ninety degrees forward and backward. The firsts lines include the libraries necessaries for the code development, the following lines call the node publishers of ROS and publishes under the names of the controllers settled in the Yaml file. The last part of the code, from $intcount = 0$; makes different positions for making the movement slow and falling-risk free.

Listing 3.18: Example control code.

```
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include <sstream>
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher joint_0_pub =
      n.advertise<std_msgs::Float64>("/mex_rb/kinect_controller0/command",
      1000);

  int count = 0;
  while (ros::ok())
  {
    std_msgs::Float64 home,msg0;
    while(count<= 20){
      msg0.data = -0.078535*count;
      ROS_INFO("Publishing: %d",count);
      joint_0_pub.publish(msg0);
      ros::spinOnce();
      loop_rate.sleep();
      ++count;
      }
   while(count>= 0){
      msg0.data = -0.078535*count;
      ROS_INFO("Publishing: %d",count);
      joint_0_pub.publish(msg0);
      ros::spinOnce();
      loop_rate.sleep();
      --count;
      }
  }
```

3.7 Launch the control files in ROS.

To run the control created for the robot, the Cpp file, it is necessary to add a few lines to the CmakeList.txt, a file that is automatically created in this chapter section 3.4.1, the CmakeList that need a modification can be found inside the project folder, the file with the same name in the src folder does

not need a modification. The Listing 3.19 shows the lines that must be added at the end of the file mentioned before.

Listing 3.19: Added lines in the CmakeList.

```
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(model_joints_publisher
    src/model_joints_publisher.cpp)
target_link_libraries(model_joints_publisher ${catkin_LIBRARIES})
```

In addition, it must be created another file with the extension *.launch* that will publish topics to the joints in the Gazebo model. The example code is shown below in the Listing 3.20, the main function of it, is to load the Joint controller configurations from Yaml file to the ROS parameter server, load the controllers, run the Joints statements, and run the Joints publisher (the Cpp file is already added in the project, in the lines added in the CmakeList file).

Listing 3.20: Example of control.launch file.

```
<?xml version="1.0"?>
<launch>

     <rosparam file="$(find mex_rb)/config/configuration.yaml"
         command="load"/>

     <node name="controller_spawner" pkg="controller_manager"
         type="spawner" respawn="false"
    output="screen" ns="/model" args="joint_state_controller
        kinect_controller0 kinect_controller? ..."/>

     <node name="robot_state_publisher" pkg="robot_state_publisher"
         type="robot_state_publisher"
    respawn="false" output="screen">
       <remap from="/joint_states" to="/model/joint_states" />
     </node>

   <node name = "model_joints_publisher" pkg ="mex_rb" type =
      "model_joints_publisher" output="screen"/>

</launch>
```

Chapter 4

Results

4.1 Gazebo Model.

To prove the correct assembly of all the Links and Joints of the robot, made in the previous chapter section 3.2, you can review in gazebo the following features.

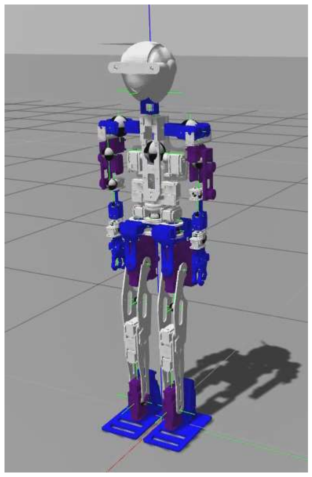


Figure 4.1: Center of mass.

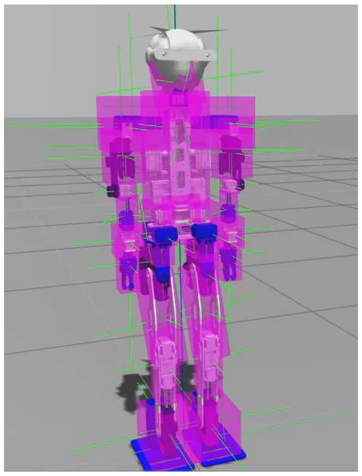


Figure 4.2: Inertial.

The information of the mass center can be verified by the activation of the property *Center of Mass*, in the menu, *View* from Gazebo. All the centers of mass must be inside the robot and in its correct center, the verification of the current robot is shown in (Figure 4.1). The (Figure 4.2) shows another

property of the same Gazebo menu, called Inertia, those are a graphical representation of the inertial forces present in the model, caused by the mass and the gravity, those inertia were set in the inertial components of the URDF, and must be near to the real robot.
The position of the joints can be reviewed in the model by the activation of the Joints propriety in the *View* menu tool, and in addition, the transparent property, to be able to easily check the Joint position in the center of each Link, the resultant robot, must show an appearance similar to the (Figure 4.3), that shows all the coordinate axis in the center of each joint. The Collisions propriety, it is similar to the other properties mentioned after in this section, the robot must have no collisions in the outlines of the model, its collision parts must be similar to the original pieces and must be placed in the same space; it can be reviewed by looking "Links" unconnected to the robot, with this property activated, those "Links" will have a transparent brown look. The (Figure 4.4) show the way a robot with all its collisions in the right places must look like.

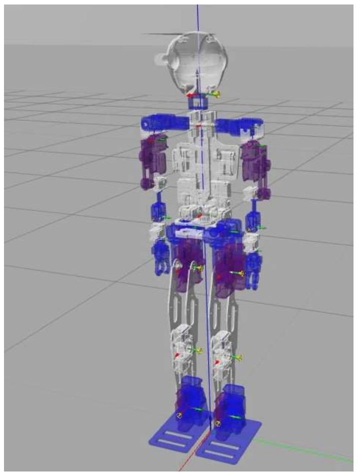


Figure 4.3: Joints position.

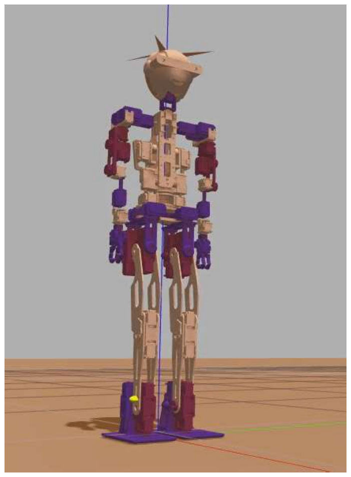


Figure 4.4: Collision Links.

To finish the verification in Gazebo, the model charged in the simulator, once the simulator is running, must fall and collide with itself while falling, and must be able to move its joints in the right directions. The robot must look like an assembled robot, its Links must not disassembly themselves of the robot while falling, or seem like rotating in a joint in the outlines of the robot.

4.2 ROS launched model.

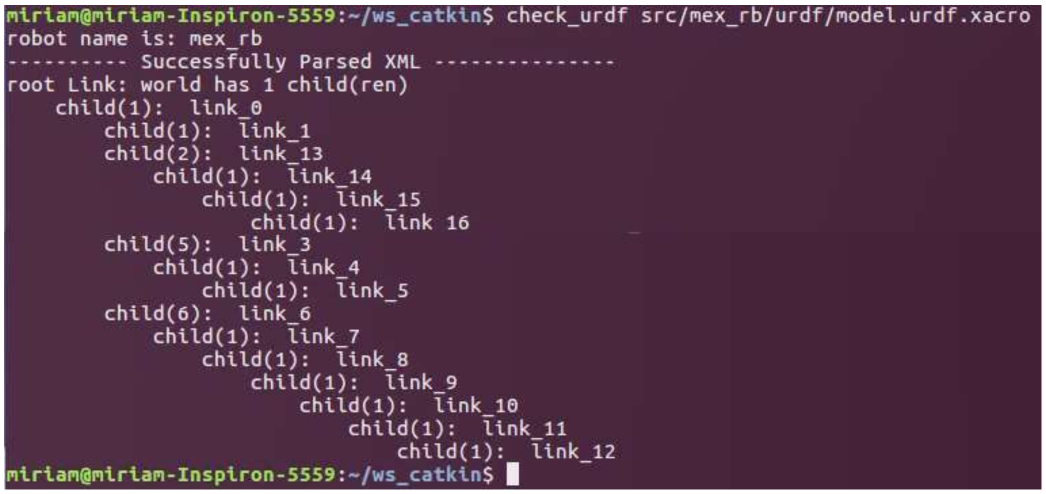
This section enumerates some shortcuts to verify the files needed to launch the robot from ROS to Gazebo and how to fix errors that arose during the elaboration of the present investigation.

4.2.1 URDF verification.

To verify that the developed URDF file that describes the robot, actually works, the easiest way founded during this investigation is to run the command "check_urdf" in the terminal. This is a ROS tool that allows the user to know if the relation parent-child is correctly formed in the *xml* structure of the URDF file. To use that command, you need to execute the command of the Listing 4.1 in the terminal, noticing that the names of the files and directories must be changed according to the desired location and name of the file to review. If the verification end successfully, the terminal will show the structure of the robot and a message as in the (Figure 4.5).

Listing 4.1: Check URDF.

```
$ check_urdf ~/folder/subfloder/subsubfolder/model.urdf
```



```
miriam@miriam-Inspiron-5559:~/ws_catkin$ check_urdf src/mex_rb/urdf/model.urdf.xacro
robot name is: mex_rb
---------- Successfully Parsed XML ---------------
root Link: world has 1 child(ren)
    child(1):  link_0
        child(1):  link_1
        child(2):  link_13
            child(1):  link_14
                child(1):  link_15
                    child(1):  link 16
        child(5):  link_3
            child(1):  link_4
                child(1):  link_5
        child(6):  link_6
            child(1):  link_7
                child(1):  link_8
                    child(1):  link_9
                        child(1):  link_10
                            child(1):  link_11
                                child(1):  link_12
miriam@miriam-Inspiron-5559:~/ws_catkin$
```

Figure 4.5: Check_urdf response.

4.2.2 Assembly mistakes.

A common error during the URDF development is to put wrong positions in the center of mass, misplace the joints or the collision. That can end up in the destruction of the model, the models with some of the error mentioned before may cause the model robot to explode, because of this, is advisable to review the properties of the *View* menu, at the beginning of this chapter 4.1. The (Figure 4.6) shows the head collision element of the model misplaced, and the (Figure 4.7) the model exploding because of a misplaced center of mass and overlarge inertias.

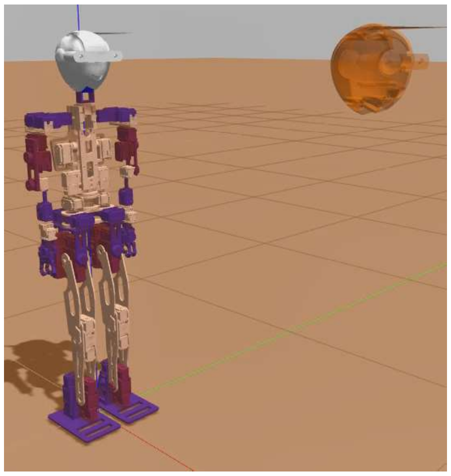


Figure 4.6: Head collision link misplaced.

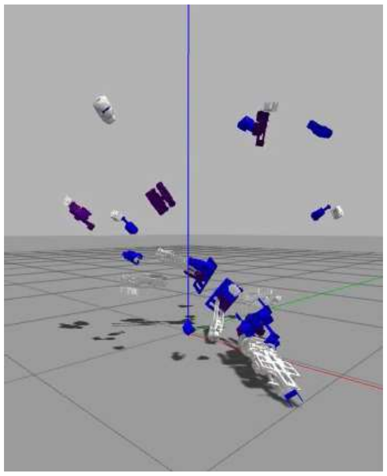
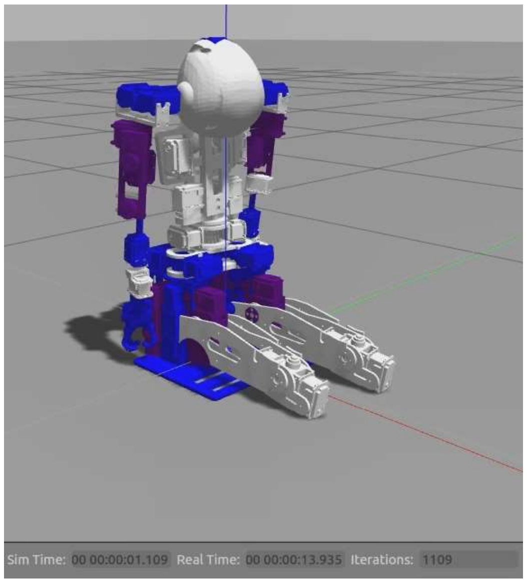


Figure 4.7: Robot dysfunctional behaviour (first model).

4.2.3 Gravity and Self Collide.

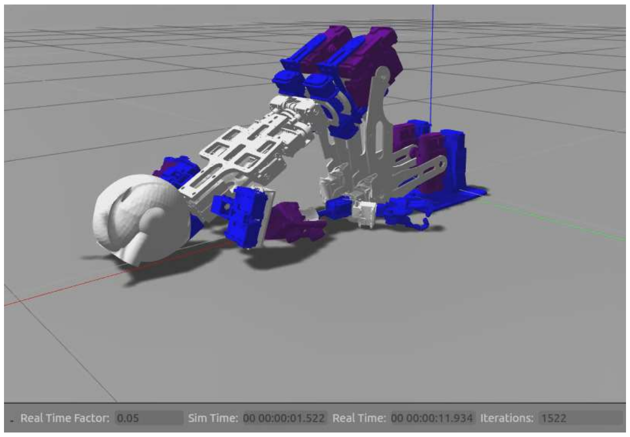
Both in the gazebo robot and in the ROS launched robot, the model must fall in a sequence analog to the real robot falling with the gravity effects, the damping of the joints and the friction between the links and in the joints. But before the verify of the gravity, it is important to assure that the *Self Collide* characteristic is active, because, other ways, the falling robot will be able to trespass itself while falling, and in the control exercises, the collisions between the links will not have an effect. An example of the robot in the simulator without self-collisions nor control it showed in the (Figure 4.8). An appropriated charge of all the physical parameters, centers of mass, joints



Sim Time: 00 00:00:01.109 Real Time: 00 00:00:13.935 Iterations: 1109

Figure 4.8: Model falling without Self collide.

and extra, resulting in the falling of the model without any control file, and eventually, the model must stop moving after reaching the floor. The simulation time must be cared of because it will show how much time is supposed to take to the robot to fall and stop moving, and based in that time, you should be able to determinate whether it was or not a logical fall. The (Figure 4.9) show the robot falling in an acceptable timing.

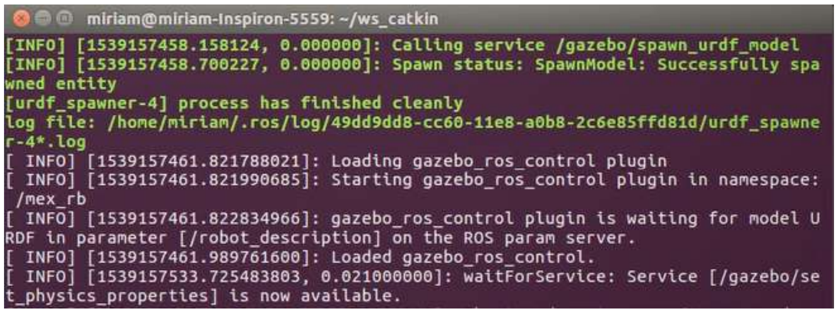


Real Time Factor: 0.05 Sim Time: 00 00:00:01.522 Real Time: 00 00:00:11.934 Iterations: 1522

Figure 4.9: Model with gravity.

4.3 ROS control.

In order to review if the robot controllers were correctly charged, the Listing 3.15 enables in its line number thirteen the *verbose* tool, which let the terminal display messages with information about the robot charge processes, such as any kind of problem during the robot and features' charge. In this point, the information that will be useful, is a message, after launching the robot with the *transmission* and *ros_ control* lines inside the URDF file and the Yaml file, the terminal must show a message as the Figure 4.10, that let the users know the processes status and if Gazebo was able to charge the control plugin and enable the physical properties with the control.



```
miriam@miriam-Inspiron-5559: ~/ws_catkin
[INFO] [1539157458.158124, 0.000000]: Calling service /gazebo/spawn_urdf_model
[INFO] [1539157458.700227, 0.000000]: Spawn status: SpawnModel: Successfully spa
wned entity
[urdf_spawner-4] process has finished cleanly
log file: /home/miriam/.ros/log/49dd9dd8-cc60-11e8-a0b8-2c6e85ffd81d/urdf_spawne
r-4*.log
[ INFO] [1539157461.821788021]: Loading gazebo_ros_control plugin
[ INFO] [1539157461.821990685]: Starting gazebo_ros_control plugin in namespace:
 /mex_rb
[ INFO] [1539157461.822834966]: gazebo_ros_control plugin is waiting for model U
RDF in parameter [/robot_description] on the ROS param server.
[ INFO] [1539157461.989761600]: Loaded gazebo_ros_control.
[ INFO] [1539157533.725483803, 0.021000000]: waitForService: Service [/gazebo/se
t_physics_properties] is now available.
```

Figure 4.10: Terminal after successfully load ROS Control.

4.3.1 Home position.

Without launch the control file, but with the controllers charged, the robot must take the "Home" position, which means that all the joints must stay put at the 0 degrees, regardless the effect of the gravity. This does not mean that the gravity is nos affecting the robot, or that the joints are not correctly working; in the action of activate the transmissions elements in the robot, is like put all the motors in the zero position and giving them the energy necessary to stay still even with the gravity and the robot mass.
This position was defined by the user at the moment of the robot development, the positions of the links and joints given in the URDF file are the ones that ROS is going to recognize as the robot's "Home" position. The terminal window must show no errors after play the simulation in Gazebo.

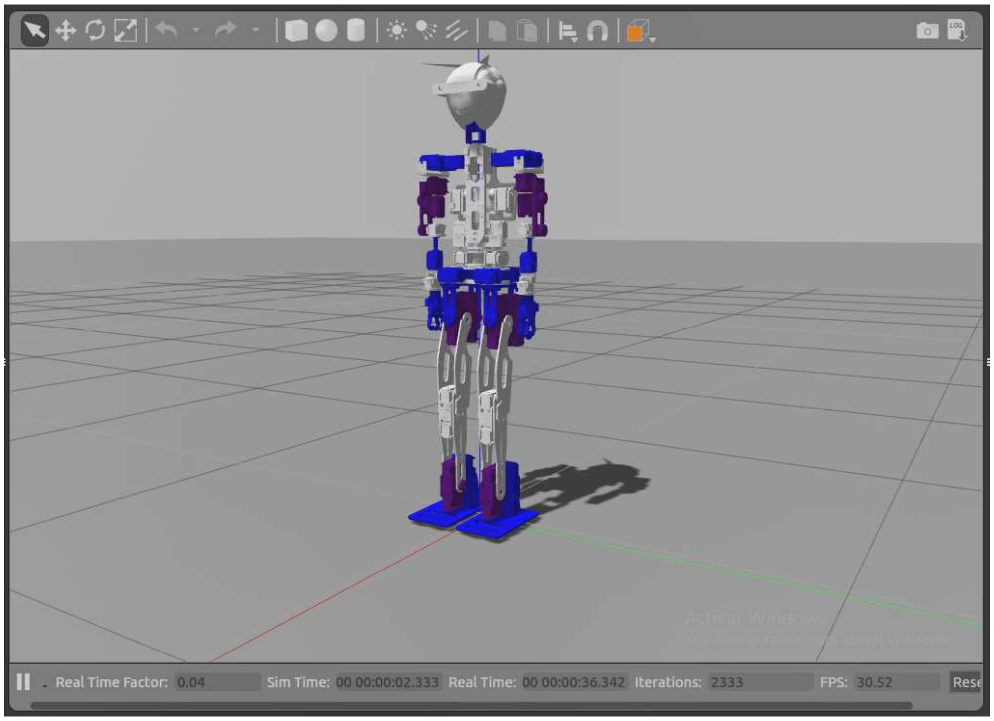

Real Time Factor: 0.04 Sim Time: 00 00:00:02.333 Real Time: 00 00:00:36.342 Iterations: 2333 FPS: 30.52

Figure 4.11: Gazebo with the model in "Home" position.

The Figure 4.11 show the simulation after a couple of second of the simulation time running, as you can see, the robot keeps the same position as before running the simulator.

If at this point, when you play the simulator, the model misbehave as in the Figure 4.7, review the PID values admitted in the Yaml file, sometimes a very small derivative gain can introduce errors during the control processes, and end up disassembling the robot and making the pieces to free "fly" in the world.

4.3.2 Arms control.

As an example of a control routine, a simple file was made to check the arms movements, and to verify its correct function. The example routine was decided to be proved only in the arms due to the complex of a walking routine, such routines are going to be developed and tested by the students of the investigation center.

Listing 4.2: Arms control example file.

```
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher trans0_pub =
      n.advertise<std_msgs::Float64>("/mex_rb/trans0_controller/command",
      1000);
  while (ros::ok())
  {
   std_msgs::Float64 msg0;
   for(int t=0; t<= 30; t++)
   {
   if ((t>0) && (t<=15))
      {
      msg0.data = 30;
      ROS_INFO("Publishing: %d",msg0);
      trans0_pub.publish(msg0);
      ros::spinOnce();
      loop_rate.sleep();
      }
   else ((t>15) && (t<=30))
      {
      msg0.data = 0;
      ROS_INFO("Publishing: %d",msg0);
      trans0_pub.publish(msg0);
      ros::spinOnce();
      loop_rate.sleep();
      }
   }
  }
  return 0;}
```

The code in the Listing 4.2 is the code for the head movements used as the Cpp file, this file puts the head and arms in movement with slow movements to avoid making the robot loss stability. The file only send instructions about the positions, the control in the motors is developed in the Yaml file, with the PID gain set for each motor, to have a better control routine, it is suggested to prove with the torque controllers, that control the force applied in each motor instead of the position that it wants to achieve.
In the Figure 4.12 a few captures of the routine are showed, in the simulator, you must see slight movements of the robot at the beginning and end of the motor routines, that is because of the force required to start any movement of the links, and the robot does not loss its stability because to the slow of the movements.

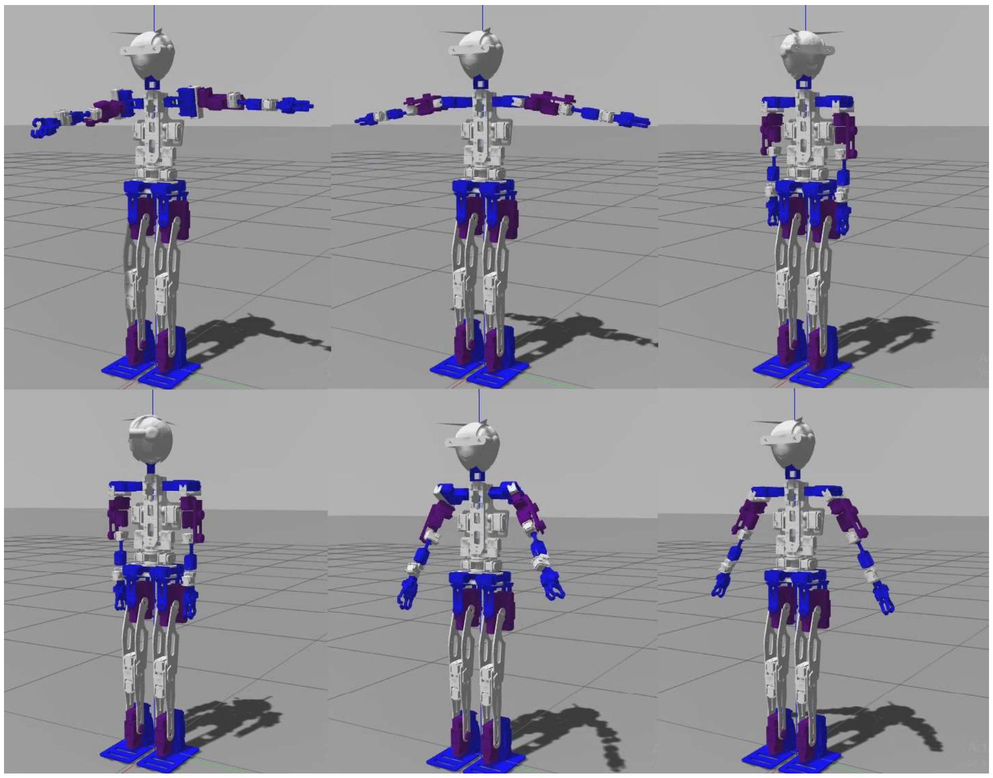


Figure 4.12: Arms control routine.

Conclusion

Finally, in the exposed work the objectives that were initially proposed have been achieved. The modeling of the humanoid robot Mex-One in the Gazebo simulator, have been created a controller for every joint of the robot, the movement, and control of the joints have been successfully achieved. The robot have been tested based in a cpp file, making able to send simple example control routines.
The project has been developed in free access software, which implies that is almost free cost, without taking account of the electrical energy, internet, and a computer, that are necessary items to develop any model, for obvious reasons. Some tips to improve the simulator development, such as use different visual and collide elements to reach a real-time factor closest to one, and some commands that will help anyone that thy to use the present document as a guide to have an easier way to verify the different steps of the work are also included.
The present work shows, it is important to have a complete knowledge of the real model that you are trying to get inside the simulator, because you need to be able to assemble the robot and create a model in the URDF file. A basic knowledge of the more common commands of Ubuntu, is necessary to have a simpler adaptation process to the ROS commands and to know how to move between them. The installations necessaries are explained in different tutorials of the software developers, from how to fragment a hard drive to how to install ROS Indigo and the specifications required of the installation machine. That was one of the best advantages founded during the elaboration of the present document, the free software has a lot of information to help to the installation and basic knowledge of it. On the other hand, one of the difficulties of the present work was that the information was very separately and the most of it was not useful or trustable, because of that, the information used in this project, was very carefully selected, trying to

show only concrete and useful facts. The present project en with a simple example of a control routine given the time limitations.
To future projects, I suggest an application called Moveit, is a ROS application to develop inverse kinematic functions, that can be employed in the control of simulated and real robots, also to change the cpp files for Python files, given that those are easier to make and, given the facilities that that language offers against the cpp files, most of the developers nowadays use Python instead of cpp.

Bibliography

[Bearnes, 2016] Bearnes, B. (2016). What's new in ubuntu 16.04. Digital Ocean, https://www.digitalocean.com/community/tutorials/what-s-new-in-ubuntu-16-04.

[Dang, 2012] Dang, D. (2012). Humanoid manipulation and locomotion with real-time footstep optimization. *Institut National Polytechnique de Toulouse*.

[Dynamics, 2018] Dynamics, B. (2018). Atlas. Boston Dynamics, https://www.bostondynamics.com/atlas.

[Foundation, 2014] Foundation, O. S. R. (2014). Beginner: Overview. GazeboSim.org, http://gazebosim.org/tutorials?tut=guided_b1&cat=.

[Honda, 2018] Honda (2018). Asimo. The power of freams, https://www.honda.mx/asimo/.

[Kato, 1986] Kato, I. (1986). *Mechanical Hands Illustrated*. Hemisphere Publishing Corporation.

[Martínez, 2008] Martínez, M. J. (2008). Sistema operativo ubuntu/linux. El Siglo de Torreón, https://www.elsiglodetorreon.com.mx/blogs/ToRo/123-sistema-operativo-ubuntu-linux.

[Mazzari, 2016] Mazzari, V. (2016). Ros - robot operating system. The Blog, https://www.generationrobots.com/blog/en/2016/03/ros-robot-operating-system-2/.

[of Engineering, 2018] of Engineering, U. C. (2018). Drc hubo. http://www.drc-hubo.com/.

[Retto, 2017] Retto, J. (2017). Sophia, first citizen robot of the world. *Universidad Nacional Mayor de San Marcos.*

[Tanaka, 2016] Tanaka, R. (2016). Make your own robot with gazebo + ros 4. make a udf file. Qiita, https://qiita.com/RyodoTanaka/items/174e82f06b10f9885265.

[Ucha, 2009] Ucha, F. (2009). Robótica. Definición ABC https://www.definicionabc.com/tecnologia/robotica.php.

[UOC, 2012] UOC (2012). La llegada de los robots humanoides. Blogs.UOC, http://informatica.blogs.uoc.edu/2012/03/08/la-llegada-de-los-robots-humanoides/.