Zagazig University, Faculty of Engineering,
Computer and Systems Engineering Dept.

ZagHexa
Design, Construction and Control of a Hexapod Walking Robot

B.Sc. Graduation Project
Submitted by

Mahmoud Mohammed Elsayed
Khaled Mohammed Risha
Mohammed Alaa Mohammed
Hend Khairy Abdelhamed
Nehad Abdelsalam Mohammed
Amira Elsayed Soliman

July, 2017

Submitted to The Computer and Systems Engineering Dept., Faculty of Engineering, Zagazig University, Egypt

Graduation Project Report to be submitted to
Zagazig University, faculty of Engineering
in partial fulfillment of the requirements for the degree
Bachelor of Science in Engineering (B.Sc.)
©2017

Date of Presentation
24. Juni 2015

Supervisors
Dr.Ing. **Mohammed Nour Abdel Gwad Ahmed**
Computer and Systems Engineering Department,
Faculty of Engineering, Zagazig University, Egypt

Defense Commete
Prof. Dr. Xyz Wuv
Prof. Dr. Abc Def

to
All, **Whom**, **we**, and **love**.
The best in my life and after ...

Abstract

This report is a documentation of the final year graduation project in electrical engineering at zagazig university. The purpose of this project is to Design, Construction and Control of a six-legged walking robot that is capable of basic mobility tasks such as walking forward, backward, rotating in place and raising or lowering the body height.

The legs are of a modular design that have three degrees of freedom each. This robot will serve as a platform onto which additional sensory components could be added, or which could be programmed to perform increasingly complex tasks.

The components that make up our final design are discussed. Also, we describe the basic robot gaits of locomotion for efficient navigation. This locomotion is tuned to make the robot faster and at same time energy efficient to navigate and negotiate difficult terrain.

The robot is an integrated multi-legged walking robot based on de-facto standard Robotic Operating System (ROS) that employs novel and different walking patterns.

Our robot is teleoperated using hand-held devices such as a smart phone or tablet or a wireless joystick. Furthermore, it has its own navigation system and a camera for instant video recording and streaming.

The power to the entire system is supplied through two 5 volts NiMH batteries. There is an additional power bank to power up the Raspberry Pi and other electronic components. We have an interactive website for robot inspection and online control in addition to leaning materials such as robot building and implementation walkthroughs and as well as step-by-setup tutorials.

Keywords – biologically inspired, legged robot, gait generation, design procedure, simulation

Acknowledgements

This graduation project consumed huge amount of work, research and dedication. Still, accomplishment would not have been possible if we did not have a support of many individuals. Therefore we would like to extend our sincere gratitude to all of them.

First of all, we would like to sincerely thank our advisors Dr.Ing. Mohammed Nour. For giving us the opportunity to work on great ideas with great people . When needing someone for the discussion of any problem, he was always there and also solved a lot of problems for/with us.

we are also grateful to our friend and Teaching Assistant, Mahmoud Ibrahim. For helping us a lot with providing the martial of the robot.

we express our warm thanks to ....., ..., and ..... for making the ..... robot available to us and the time they spend assisting us to carry out the field experiments. Without their superior knowledge and experience, the experiments would not have that like in quality of outcomes, and thus their support has been essential. In addition, we wish to express our sincere gratitude to ..... for helping when we had questions as well as frustrations.

We would like to thank all the numerous people in the internet who ask questions and provide answers for programming problems (specifically in ROS) and for the very useful code and tools they share with others.

we would like to most importantly acknowledge the effort of our families, who encouraged us to pursue higher education and support us through the difficulties associated with such a goal even when we was not sure we would make it through.

Last but not least, we would like to thank our friends for always encouraging us onwards. We can never thank them enough for their love and faith.

This work was supported by a financial aid from Zagazig University. We would like to thank the funders. They had no role in study design, data collection and analysis, decision to publish, or preparation of this work.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols and Abbreviations

DOF	Degree of freedom
PWM	Pulse Width Modulation
I^2C	Inter-Integrated Circuit
Hexapod	six-leg walking robot
LTS	Long Term Support
LiPo	Lithium Polymer
RPi	Raspberry Pi
GPIO	General-purpose input/output
CLI	Command Line Interface
LPF	Low Path Filter
HPF	High Path Filter
FPS	Frame Per Seconds
GND	Ground

"There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success than to take the lead in the introduction of a new order of things."

— Niccolo Machiavelli, (Italian writer and statesman, Florentine patriot, author of 'The Prince', 1469-1527)

Chapter 1
Introduction

«««< HEAD

======= »»»> origin/master In today's technological society, people have grown accustomed to daily use of several kinds of technology from personal computers to supercomputers, from personal vehicles to commercial airplanes, from mobile phones to communicating through the Internet and everything in between. Robotics technology has been a hot topic recent years. As such, the use of robots has become increasingly common. As robots can be used to complete repeated tasks, increase manufacturing production, carry extra weight and many other common tasks that humans do. Therefore, robots can be found everywhere. So far, all mobile robots used in extraterrestrial surface exploration missions were wheel-driven systems. However, even if such a system is equipped with a suspension system, the capability to surmount obstacles and to conquer steep inclinations is limited. Also driving on fine-grained soil can become a problem for these kind of systems. Multi-legged walking systems, in contrast, are equipped with a highly flexible locomotor system. Along with appropriate control strategies it should offer them the capability to securely maneuver in rough and steep environments. Major counter arguments for legged systems are the higher complexity regarding the mechanical design and control as well as the comparatively high power consumption. Thus, the challenge lies in minimizing these drawbacks and in exploiting the potentialities of such systems. One of the most important part of a robot is its chassis. There are several basic chassis types: wheeled, tracked and legged chassis. Wheeled chassis are fast, but not suitable for rough terrain. Tracked chassis are slower, but more suitable to rugged terrain. Legged chassis are quite slow and more difficult to control, but extremely robust in rough terrain. Legged chassis are capable to cross-large holes and can operate even after losing a leg [Saranlı, 2002]. Extensive research is conducted in this field because of its large potential. Legged chassis are especially ideal for space missions [terrain hex-limbed extra-terrestrial explorer, , Tedeschi and Carbone, 2014] . There are also several projects in military research [Dynamics, 2015b, Dynamics, 2015a].

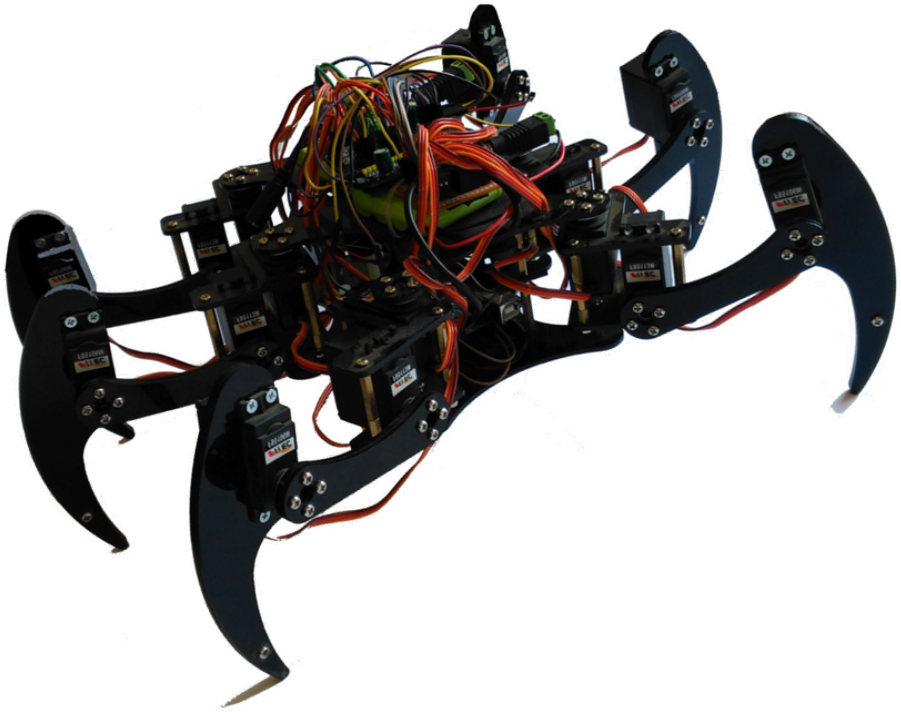

Figure 1.1.: A sex-legged walking robot.

One of the interesting features of hexapod robots such as our ZagHexa (shown in Fig.Figure 1.1) is that they can climb over obstacles larger than the equivalent sized wheeled or trucked vehicle. In fact, the use of wheels or crawlers limits the size of the obstacle that can be climbed to half the diameter of the wheels. On the contrary, legged robots can overcome obstacles that are comparable with the size of the machine leg[terrain hex-limbed extra-terrestrial explorer,]. Hexapod walking robots also benefit from a lower impact on the terrain and have greater mobility in natural surroundings. This is especially important in dangerous environments like mine fields, or where it is essential to keep the terrain largely undisturbed for scientific reasons [Tedeschi and Carbone, 2014].

Hexapod legged robots have been used in exploration of remote locations and hostile environments such as seabed [Dynamics, 2015b], in space or on planets [Dynamics, 2015a, Moore and Buehler, 2001] in nuclear power stations [Ding et al., 2010], and in search and rescue operations[Manoiu-Olaru et al., 2011]. Beyond this type of application, hexapod walking vehicles can also be used in a wide variety of tasks such as forests harvesting, in aid to humans in the transport of cargo, as service robots and entertainment. Development of hexapods is increasingly robust in the military sector. Armies all over the world are exploring ways of using hexapods to detect land mines, traverse rocky, unstable terrain, and carry out simple delivery missions in danger zones.

1.1. History

Robots inspired by insects and other animals have previously been designed with physical antennae and tactile sensors to navigate their environment, as in the work by Brooks (1989) [**?**, **?**], Cowan et al. (2005), Hartmann (2001) [**?**] and Lee et al. (2008) [Digia, 2017]; the last three works employed the use of a single tactile element rather than a pair [MohdDaud and KenzoNonami, 2012]. Because of their extreme mobility and agile adaptability to irregular terrain, insects have long been an inspiration for the designers of mobile and legged robots [Lewinger and Quinn, 2010, Lewinger and MartinReekie, 2011]. Early hexapod robots such as Genghis and later creations such as Tarry implemented insect-like mobility based on observations of insect behaviors. The inter- leg coordination system developed by Holk Cruse [**?**, **?**] has been widely implemented in legged hexapods and its basis is in behavioral experiments that qualitatively analyzed insect walking behaviors [Dürr et al., 2004].

1.1.1. Early Designs

The first hexapods can be identified as robots based on a rigidly predetermined motion so that an adaptation to the ground was not possible. Early researches in the 1950s were focused on assigning the motion control completely by a human operator manually[Schneider and Schmucker, 2006].

One of the first successful hexapod robot was constructed at University of Rome in 1972 (Fig.Figure 1.2a) as a computer-controlled walking machine with electric drives[Peternella and Salinari, 1973]. In the middle 70s, at the Russian Academy of Sciences in Moscow, a six-legged walking machine was developed with a mathematical model of motion control. It was equipped with a laser scanning range finder and was connected with a two-computer control system [Okhotsimski and Platonov, 1973]. In 1976, Masha hexapod walking robot was designed at Moscow State University (Fig.Figure 1.2b). The robot had a tubular axial chassis, articulated legs with three DoFs [Gurfinkel et al.,]. The hexapod was able to negotiate obstacles using contact on the feet and a proximity sensor. Ohio State University in 1977 developed a six-legged insect-like robot system called “OSU Hexapod” [McGhee, 1977]. This hexapod was kept tethered and was made to walk short distances over obstacles (Fig.Figure 1.2c). In 1984, Odetic Inc., California, USA, developed Odex I [Byrd and de Vries, 1990], a six-legged radially symmetric hexapod robot which used an onboard computer to play back pre-programmed motions (Fig.Figure 1.2d).

1.1.2. Recent Developments

The two last decades have been characterized by a rapid development of control systems technology. Hexapod robots were equipped with various sensing systems. Artificial

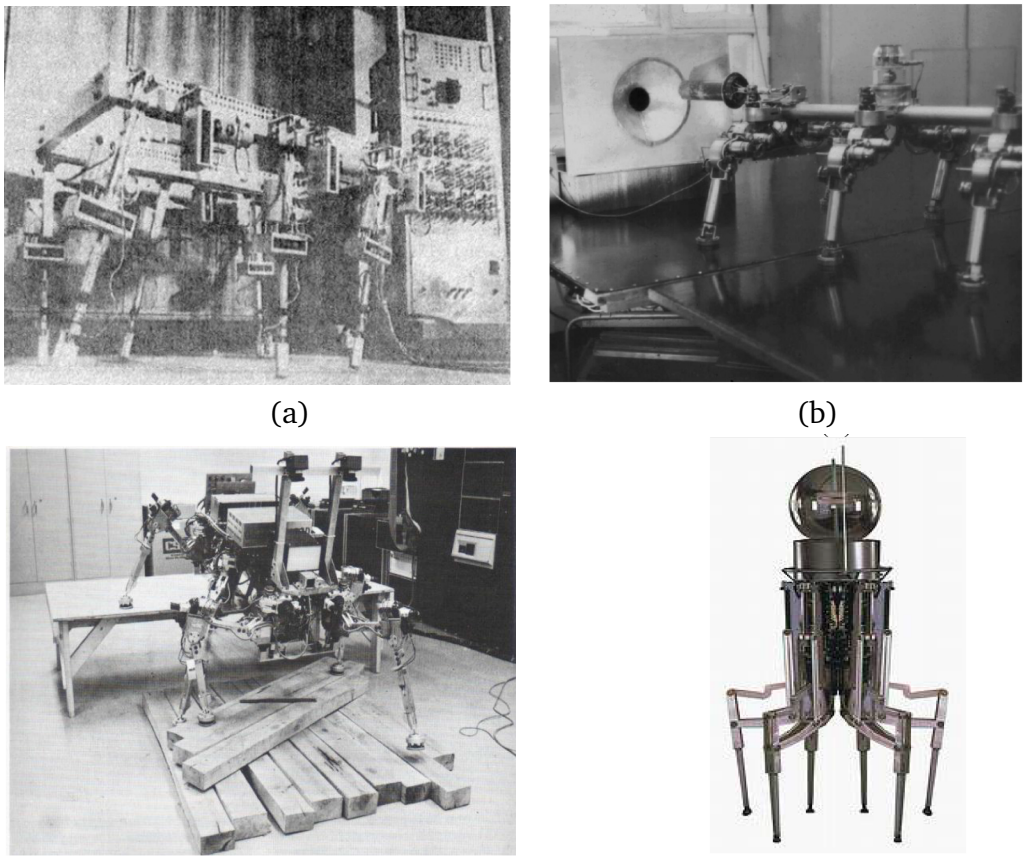

Figure 1.2.: Early hexapod design: (a) University of Rome's hexapod; (b) MASHA hexapod; (c) OSU hexapod; (d) ODEX I hexapod.

Intelligence systems were widely applied to the analysis of environment and motion of robots on a complex surface. A series of bio inspired robots was developed at Case Western Reserve University (USA) at the end the 90s, such as, for example, Robot III that had a total of 24 DoFs. Robot III architecture was based on the structure of cockroach, trying to imitate their behavior [**?**]. In particular, each rear leg had three DoFs, each middle leg four DoFs and each front leg five DoFs. Similarly, Biobot was a biomimetic robot physically modeled as the American cockroach (Periplaneta Americana) and powered by pressurized air [**?**]. This hexapod had a great speed and agility.
Each leg of the robot had three segments, corresponding to the three main segments of insect legs: coxa, femur, and tibia.

Figure 1.3.: Some Example on recent developments in hexapod design

"It is impossible for us, who live in the latter ages of the world, to make observations in criticism, morality, or in any art or science, which have not been touched upon by others. We have little else left us but to represent the common sense of mankind in more strong, more beautiful, or more uncommon lights."

— Joseph Addison, (English essayist, poet, and politician, 1672–1719), Spectator, No. 253

Chapter 2
Design Considerations

In this chapter we will walk through the hardware design of our hexapod including both mechanical and electronic parts

2.1. Design consideration

Designing hexapod legged robots is far from trivial. A very numerous and a wide range of possibilities exist to design a hexapod. Designers must take several decisions which influence the operation and technical features. Some of the most important design issues and constraints according to [**?**] can be outlined as:

- The mechanical structure of robot body.
- Leg architecture.
- Max sizes.
- Actuators and drive mechanisms.
- Control architecture.
- Power supply.
- Walking gaits and speed.
- Obstacle avoidance capability.
- Payload.
- Autonomy.
- Operation features.
- Cost.

The above mentioned design issues and constraints can be classified as design input (or key features) and design output (or main design characteristics) as shown in the scheme of Figure 2.1.

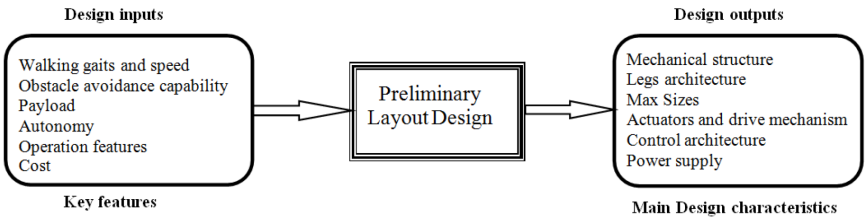

Figure 2.1.: A scheme for preliminary layout design of hexapod walking robots.

2.2. Hardware overview

ZagHexa is a hexapod robot with 18 DOFs (three degrees of freedom (DOF) for each leg), it can walk in any direction (translation), or turn in place (rotation), or any combination of the two. The leg lift and ride height is adjustable as well. The robot uses a distributed walking control system based on the neurobiology of insects, stepping in the sagittal plane to angled stepping, which then induces turning in the robot. It is an integrated multi-legged walking robot based on de-facto standard Robotic Operating System (ROS) that employs novel and different walking patterns. Our robot is teleoperated using hand-held devices such as a smart phone or tablet or a wireless joystick (see Fig.Figure 2.2). Furthermore, it has its own navigation system and a camera for instant video recording and streaming. The power to the entire system is supplied through two 5 volts NiMH batteries. There is an additional power bank to power up the Raspberry Pi and other electronic components.

The final design of the ZagHexa robot constructed mainly with acrylic is shown in Fig.Figure 1.1. ZagHexa body moves independently of its ground contact points. To make its center of gravity shift on a horizontal plane, forward/backward, and sideways moving functions are effective. These functions can also produce a smooth body movement independently of intermittent leg traveling. The robot has been designed with three degrees of freedom in the front, middle and rear legs respectively. The physical specifications are given in Table 2.

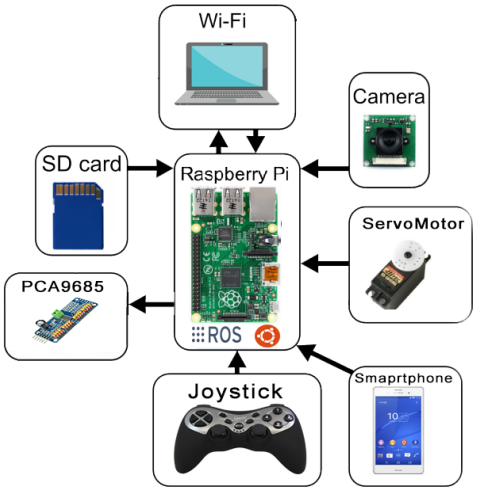

Figure 2.2.: The electronic system of the robot.

Parameter	Description
Length	30cm
Width	27cm
Height	17cm
Weight	3Kg
Construction Material	
Actuators	DC servo motors
Motion Control	Servo Sequential Control
Leg Stroke (Max)	6cm
Leg Lift (Max)	5cm

2.3. Mechanical design

In this section we will discuss the different designs of the robot including early ones as well as the final one.

2.3.1. Micro ZagHexa

We started by making a small hexapod with 18 micro servos to test our basic functions and walking algorithms. The basic parts of Micro ZagHexa is made of a 1 mm thick Aluminium sheet.

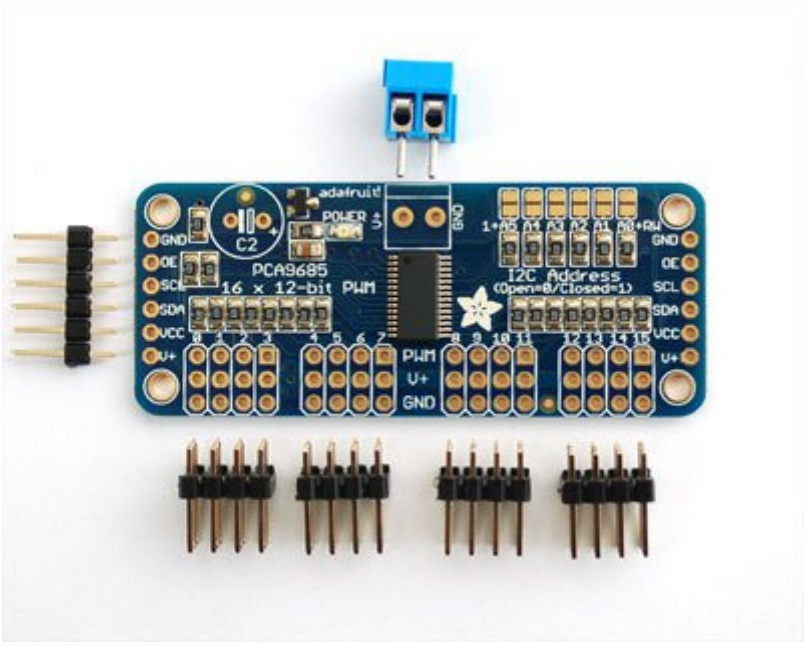

Figure 2.3.: Micro ZagHexa

2.3.2. ZagHexa

We moved to our next hexapod design and made it bigger and more natural looking with an articulate leg and body design. ZagHexa is made of 2.5 mm thick Acrylic. **??** shows the CAD model of Zaghexa and **??** shows it after being assembled.

2.3.3. Giant ZagHexa

Lastly, we made the giant version of our hexapod. It is made of a double layer 1 mm thick Aluminium sheet. The CAD model of Giant ZagHexa is shown in **??** and Figure 2.7 shows it after assembling. The giant ZagHexa was a little disappointing as servo motors were not strong enough to drive our hexapod. We tried to make the chassis lighter but that did not solve the problem completely. It still have stability issues.

2.4. Electronics

In this section we will talk about the electronic components we have been using through the whole project.

Figure 2.4.: ZagHexa during assembling

2.4.1. Adafruit PCA9685

Firstly, we used the Adafruit PCA9685 servo controller shown in Figure 2.8 and **??**. Although driving servo motors with the Arduino Servo library is pretty easy, each one consumes a precious pin - not to mention some Arduino processing power. The Adafruit 16-Channel 12-bit PWM/Servo Driver will drive up to 16 servos over I2C with only 2 pins. The on-board PWM controller will drive all 16 channels simultaneously with no additional Arduino processing overhead. Moreover, we can chain up to 62 of them to control up to 992 servos - all with the same 2 pins!

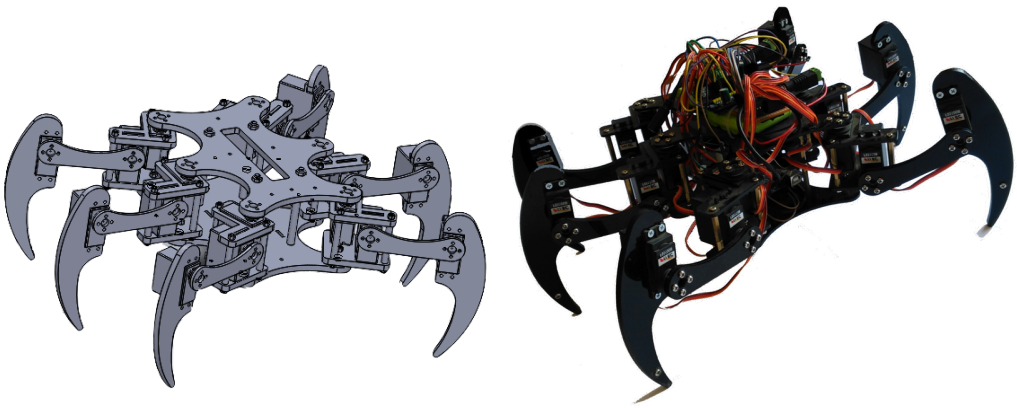

Figure 2.5.: CAD model (left) and assembled ZagHexa (right)

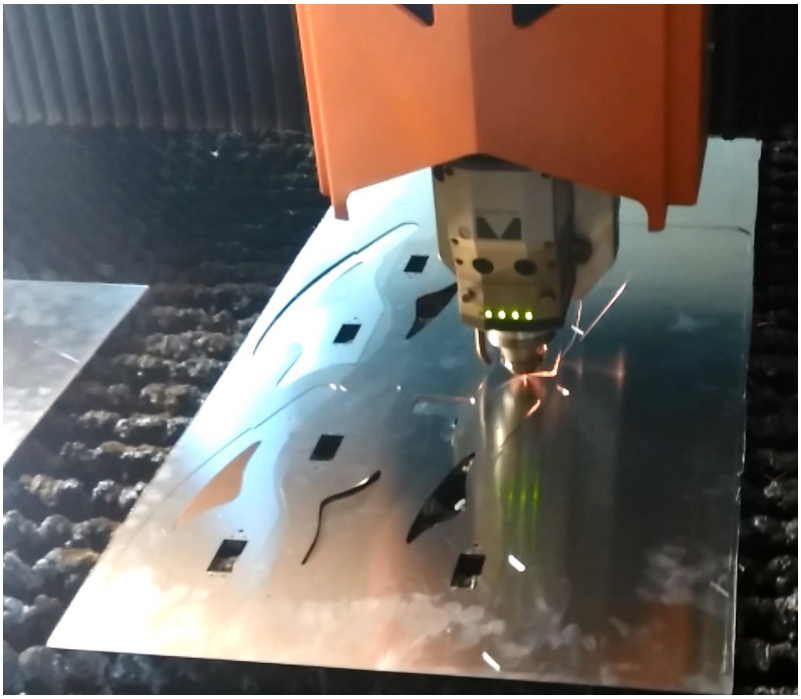

Figure 2.6.: Giant ZagHexa during laser cutting

Figure 2.7.: CAD model (left) and assembled Giant ZagHexa (right)

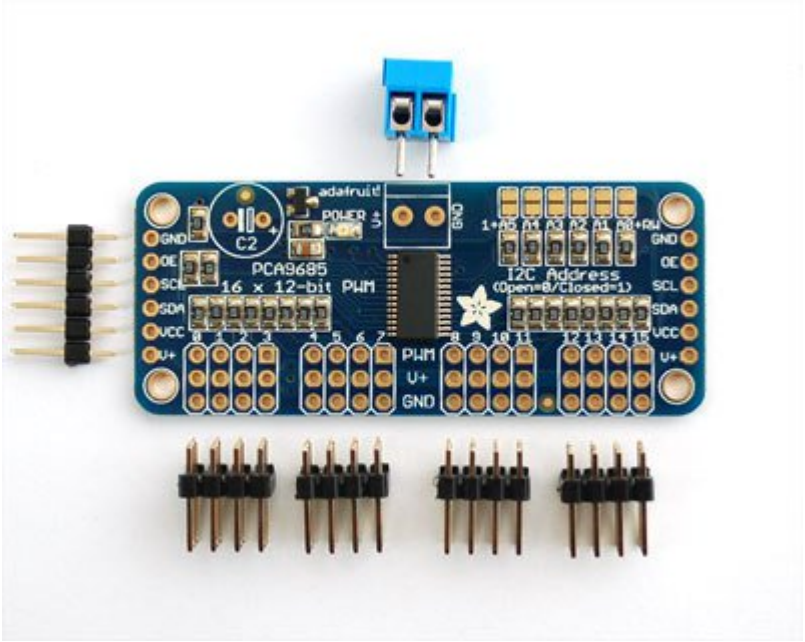

Figure 2.8.: Adafruit PCA9685

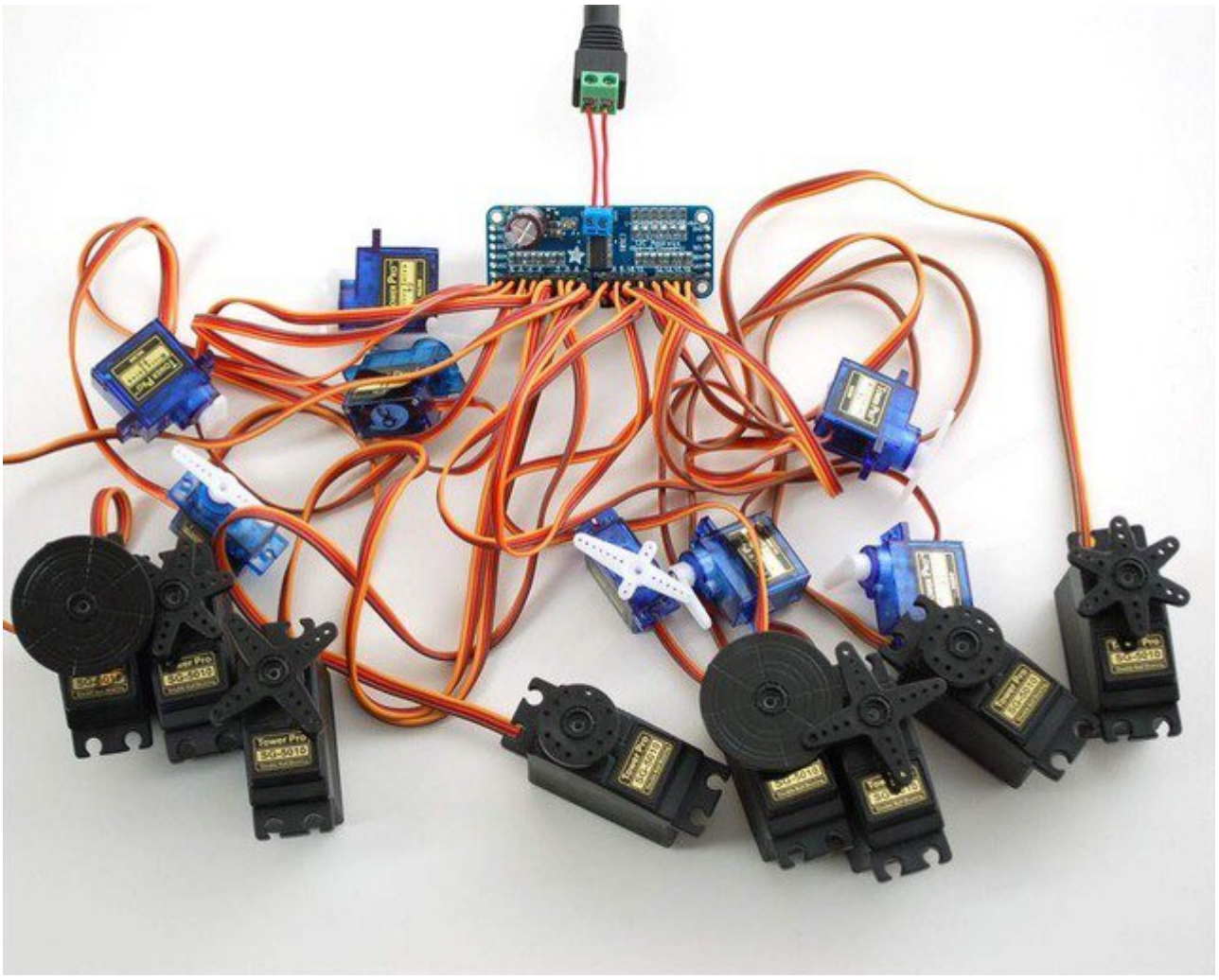

Figure 2.9.: Adafruit PCA9685 wiring

Chapter 3
Simulation

Programming directly on a real robot gives us good feedback and it is more impressive than simulations, but not everybody has possible access to real robots. For this reason, we have programs that simulate the physical world.
The first phase of robot manufacturing is its design and modeling. We can design and model the robot using CAD tools such as Solid Works, Blender, and so on. One of the main purposes of modeling robot is simulation. The robotic simulation tool can check the critical flaws in the robot design and can confirm the working of the robot before it goes to the manufacturing phase.

The virtual robot model must have all the characteristics of real hardware, the shape of robot may or may not look like the actual robot but it must be an abstract, which has all the physical characteristics of the actual robot.

If we are planning to create the 3D model of the robot and simulate using ROS, you need to learn about some ROS packages which helps in robot designing. ROS has a standard meta package for designing, and creating robot models called robot model, which consists of a set of packages called urdf, robot state publisher and so on. These packages help us create the 3D robot model description with the exact characteristics of the real hardware.

In this chapter, we will cover the following topics:

1. ROS packages for robot modeling
2. Understanding robot modeling using URDF
3. Creating our URDF model
4. Watching the 3d model in RVIZ
5. Making our robot movable

3.1. ROS packages for robot modeling

The way ROS uses the 3D model of a robot or its parts, to simulate them. ROS provides some good packages that can be used to build 3D robot models.
In this section, we will discuss some of the important ROS packages that are commonly used to build robot models:

robot model: ROS has a meta package called robot model, which contains important packages that help build the 3D robot models. We can see all the important packages inside this meta- package:

URDF: One of the important packages inside the robot model meta package is urdf. The URDF package contains a C++ parser for the Unified Robot Description Format (URDF), which is an XML file to represent a robot model.

We can define a robot model, sensors, and a working environment using URDF and can parse it using URDF parsers.
We can only describe a robot in URDF that has a tree-like structure in its links, that is, the robot will have rigid links and will be connected using joints. Flexible links can't be represented using URDF.
The URDF is composed using special XML tags and we can parse these XML tags using parser programs for further processing. We can work on URDF modeling in the upcoming sections.

joint state publisher: This tool is very useful while designing robot models using URDF.
This package contains a node called joint state publisher, which reads the robot model description, finds all joints, and publishes joint values to all non fixed joints using GUI sliders.
The user can interact with each robot joint using this tool and can visualize using RViz. While designing URDF, the user can verify the rotation and translation of each joint using this tool.

kdl parser: Kinematic and Dynamics Library (KDL) is an ROS package that contains parser tools to build a KDL tree from the URDF representation. The kinematic tree can be used to publish the joint states and also to forward and inverse kinematics of the robot.

robot state publisher: This package reads the current robot joint states and publishes the 3D poses of each robot link using the kinematics tree build from the URDF. The 3D pose of the robot is published as ROS tf (transform). ROS tf publishes the relationship between coordinates frames of a robot.

xacro: Xacro stands for (XML Macros) and we can define how xacro is equal to URDF plus add-ons. It contains some add-ons to make URDF shorter, readable, and can be used for building complex robot descriptions. We can convert xacro to URDF at any time using some ROS tools. We will see more about xacro and its usage in the upcoming sections.

3.2. Understanding robot modeling using URDF

We have discussed the urdf package. In this section, we will look further at the URDF XML tags, which help to model the robot. We have to create a file and write the relationship between each link and joint in the robot and save the file with the .urdf extension.
The URDF can represent the kinematic and dynamic description of the robot, visual representation of the robot, and the collision model of the robot.

The following tags are the commonly used URDF tags to compose a URDF robot model:
link: The link tag represents a single link of a robot. Using this tag, we can model a robot link and its properties. The modeling includes size, shape, color, and can even import a 3D mesh to represent the robot link. We can also provide dynamic properties of the link such as inertial matrix and collision properties.
The syntax is as follows:

```
<link name="<name of the link>">
    <inertial>...........</inertial>
    <visual> ............</visual>
    <collision>..........</collision>
</link>
```

The following is a representation of a single link. The Visual section represents the real link of the robot, and the area surrounding the real link is the Collision section. The Collision section encapsulates the real link to detect collision before hitting the real link.
joint: The joint tag represents a robot joint. We can specify the kinematics and dynamics of the joint and also set the limits of the joint movement and its velocity. The joint tag supports the different types of joints such as revolute, continuous, prismatic,fixed, floating, and planar.

The syntax is as follows:

```
<joint name="<name of the joint>">
  <parent link="link1"/>
  <child link="link2"/>
  <calibration .... />
  <dynamics damping ..../>
  <limit effort .... />
</joint>
```

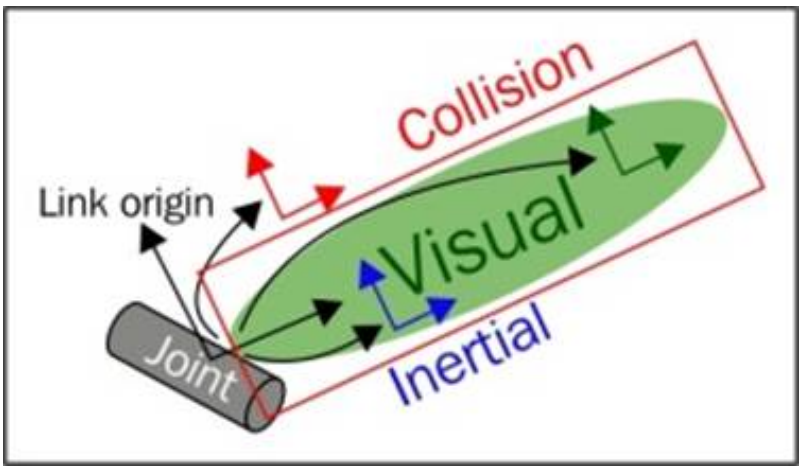

Figure 3.1.: Visualization of a URDF link

A URDF joint is formed between two links; the first is called the Parent link and the second is the Child link. The following is an illustration of a joint and its link:
robot: This tag encapsulates the entire robot model that can be represented using

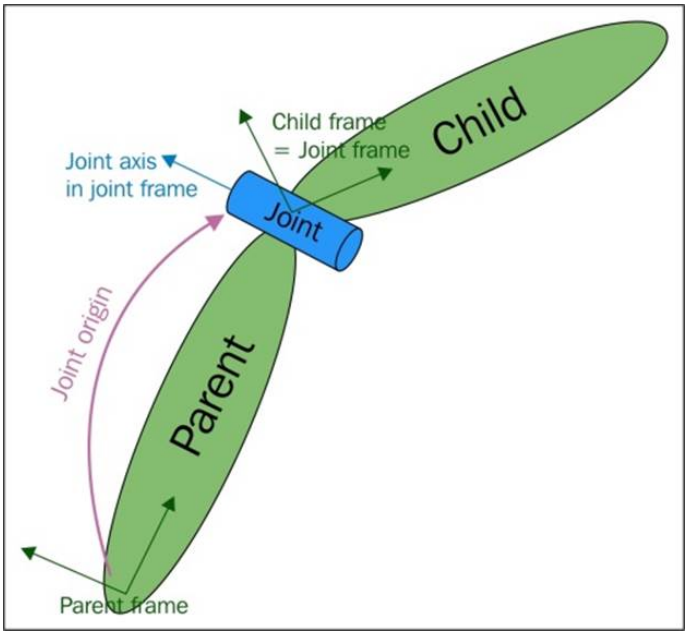

Figure 3.2.: Visualization of a URDF joint

URDF. Inside the robot tag, we can define the name of the robot, the links, and the joints of the robot. The syntax is as follows:

```
<robot name="<name of the robot>"
  <link> ..... </link>
  <link> ..... </link>
  <joint> ..... </joint>
```

```
  <joint> ..... </joint>
</robot>
```

A robot model consists of connected links and joints. Here is a visualization of the robot model:

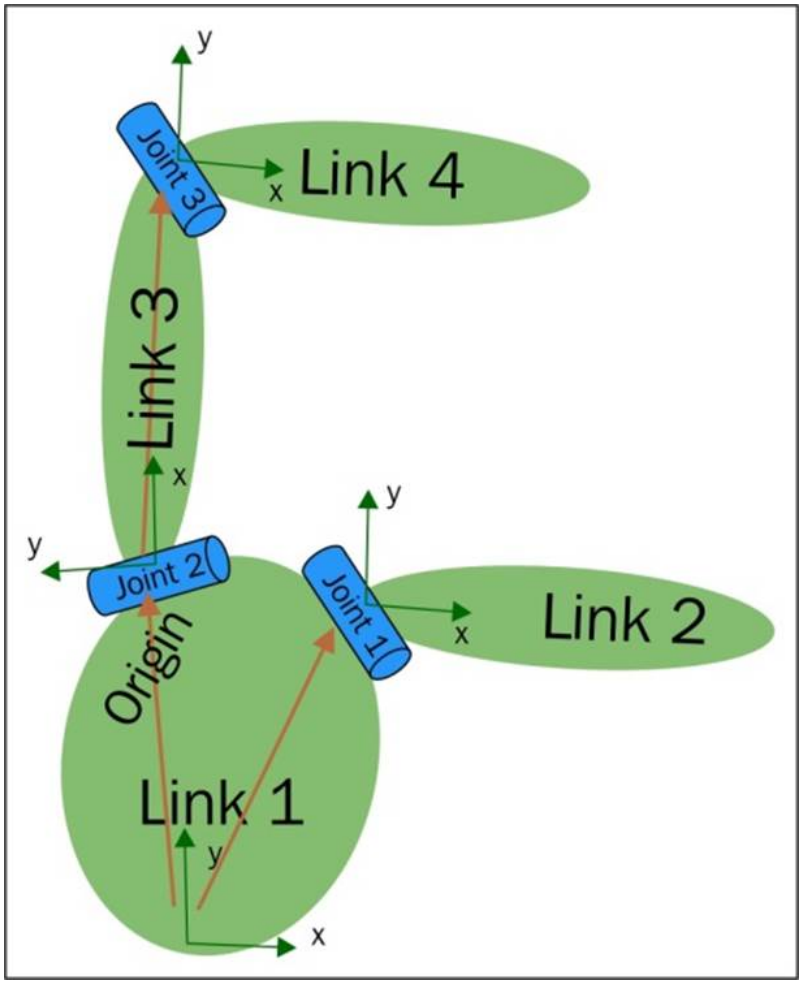

Figure 3.3.: Visualization of a robot model having joints and links

3.3. Creating our URDF model

To start first we create the urdf file, let's call it *zaghexasim.urdf* and put in the following code; this URDF code is based on XML.As you will see in the code, there are two principal fields that describe the geometry of a robot: links and joints.
the first link has the name base link; this name must be unique to the file

```
<?xml version="1.0" ?>
<robot name="zaghexa" xmlns:xacro="http://ros.org/wiki/xacro">
 <!-- Build the body frame -->
<link name="base_link"/>
<joint name="base_joint" type="fixed">
<parent link="base_link"/>
<child link="box"/>
<origin rpy="0 0 0" xyz="0 0 0"/>
</joint>
<link name="box">
<visual>
<origin rpy="0 0 0" xyz="0 0 0"/>
<geometry>
<mesh filename="package://zaghexa_sim/meshes/box.STL"/>
</geometry>
<material name="grey">
<color rgba="0.5 0.5 0.5 1"/>
</material>
</visual>
</link>
```

In the joint field we define the name which must be unique as well also we define the type of joint(fixed,revolute,continous,floating or planar) the parent, and the child. in our case tibia,femur and leg centre joint are the children of base link which is fixed but all of other joints are revolute.
this is a sample of one leg and how does it build

```
<!-- Joint properties -->
<!-- Leg macros -->
<!-- Build robot model -->
<joint name="leg_center_joint_r1" type="fixed">
<origin rpy="0 0 0" xyz="0.087598 -0.050575 0"/>
<parent link="box"/>
<child link="leg_center_r1"/>
</joint>
<link name="leg_center_r1"/>
<joint name="coxa_joint_r1" type="revolute">
<origin rpy="0 0 -1.0471975512" xyz="0 0 0"/>
<parent link="leg_center_r1"/>
<child link="coxa_r1"/>
<axis xyz="0 0 -1"/>
<limit effort="10000" lower="-1.5" upper="1.5" velocity="100"/>
</joint>
<link name="coxa_r1">
<visual>
<origin rpy="0 0 0" xyz="0 0 0"/>
<geometry>
<mesh filename="package://zaghexa_sim/meshes/coxa_r.STL"/>
</geometry>
<material name="">
<color rgba="0.7 0.7 0 1"/>
```

```
</material>
</visual>
</link>
<joint name="femur_joint_r1" type="revolute">
<origin rpy="-1.57079632679 0 0" xyz="0.0294 0 0"/>
<parent link="coxa_r1"/>
<child link="zaghexa"/>
<axis xyz="0 0 -1"/>
<limit effort="10000" lower="-1.5" upper="1.5" velocity="100"/>
</joint>
<link name="zaghexa">
<visual>
<origin rpy="0 0 0" xyz="0 0 0"/>
<geometry>
<mesh filename="package://zaghexa_sim/meshes/femur_r.STL"/>
</geometry>
<material name="">
<color rgba="0 0.7 0.7 1"/>
</material>
</visual>
</link>
<joint name="tibia_joint_r1" type="revolute">
<origin rpy="3.14159265359 0 1.57079632679" xyz="0.08 0 0"/>
<parent link="zaghexa"/>
<child link="tibia_r1"/>
<axis xyz="0 0 1"/>
<limit effort="10000" lower="-1.5" upper="1.5" velocity="100"/>
</joint>
<link name="tibia_r1">
```

You can check the syntax of the urdf whether we have errors, we can use: check urdf command tool:

```
$ rosrun urdf_parser check_urdf zaghexa_sim.urdf
```

If you want to see it graphically, you can use the urdf to graphiz command tool

```
$ rosrun urdf_parser urdf_to_graphiz "`rospack find zaghexa_sim`/
    urdf/zaghexa_sim.urdf"
```

The following is what you will receive as output:

3.4. Watching the 3D model in RVIZ

Now that we have the model of our robot, we can use it on rviz to watch it in 3D and see the movements of the joints.
We will create the display.launch file in zaghexa-sim/launch folder, and put the following code in it:

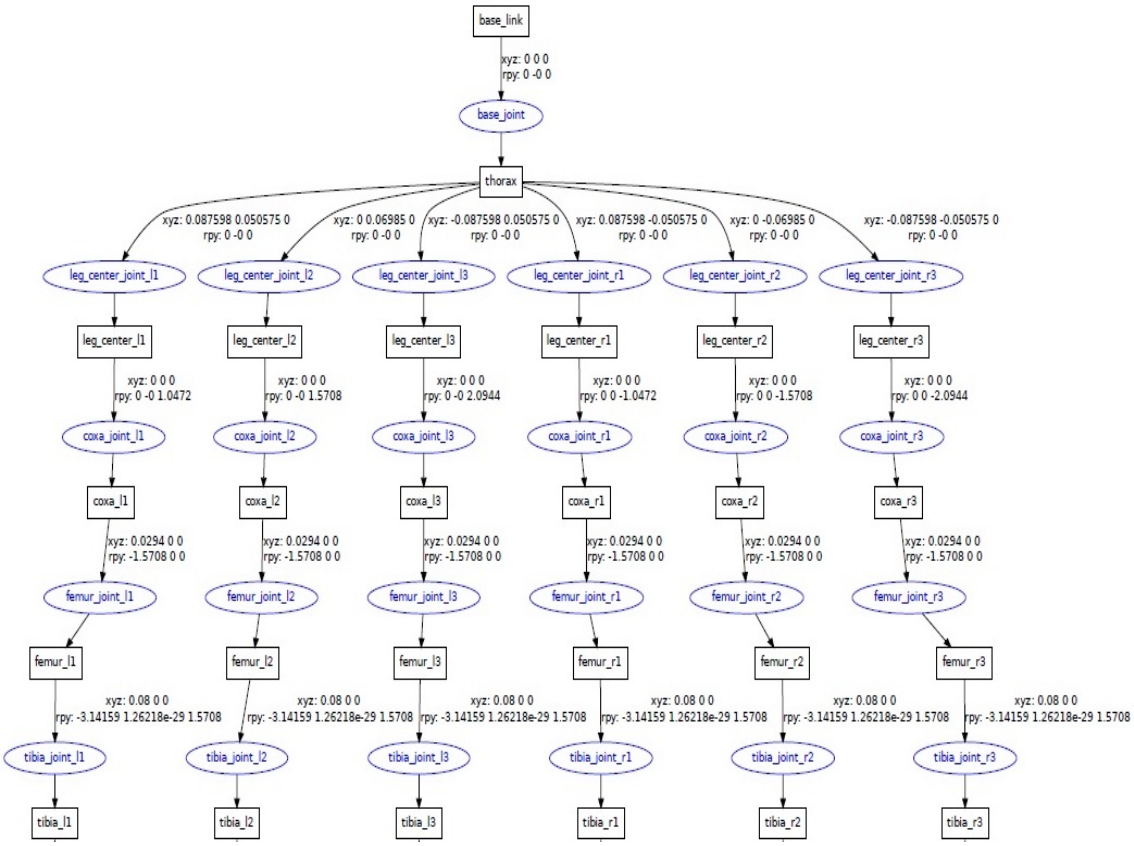

Figure 3.4.: output of urdf to graphics

```
>>>>>>> origin/master
        <launch>
        <arg
        name="model" />
        <arg
        name="gui"
        default="True" />
        <param
        name="robot_description"
        command="$(find xacro)/xacro.py '$(find zaghexa_sim)/models/zaghexa_model.
            xacro'" />
        <param
        name="use_gui"
        value="$(arg gui)" />
        <param
        name="rate"
        value="25" />
        <rosparam param="source_list">
        [leg_joints_states]
        </rosparam>
        <node
```

```
        name="joint_state_publisher"
        pkg="joint_state_publisher"
        type="joint_state_publisher" />
        <node
        name="robot_state_publisher"
        pkg="robot_state_publisher"
        type="state_publisher" />
        <node
        name="rviz"
        pkg="rviz"
        type="rviz"
        args="-d $(find zaghexa_sim)/urdf.rviz" />
        </launch>
<<<<<<< HEAD
```

We will launch it with the following command:

```
$ roslaunch zaghexa_sim display_model.launch model:="‘
    rospack find
zaghexa_sim‘/urdf/zaghexa_sim.urdf"
```

if every thing is fine and you have no errors, it will load RVIZ and you will see:

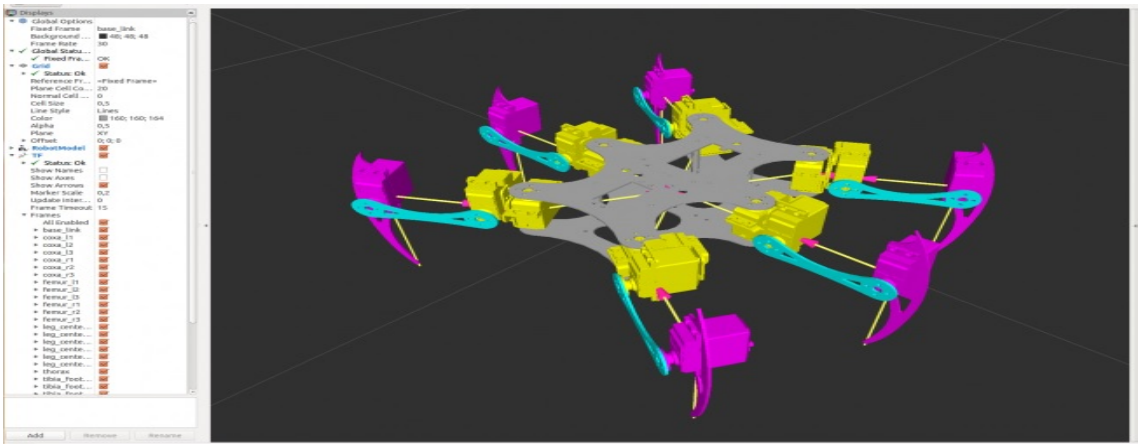

Figure 3.5.: output of urdf to graphics

3.4.1. Making our robot movable

A good way of testing whether or not the axis and limits of the joints are fine by running rviz with joint state publisher GUI

```
$ roslaunch zaghexa_sim display.launch model:="‘rospack
    find
zaghexa_sim‘/urdf/zaghexa_sim.urdf" gui:=true
```

you will see a GUI with some sliders each of them controls one joint of the 18 joints so we have 18 sliders:

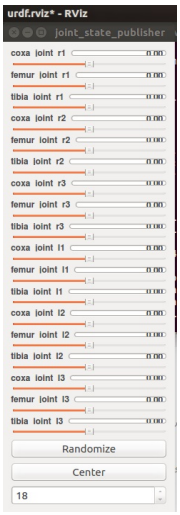

Figure 3.6.: Joint state publisher GUI

In the next figures you will see the effect of changing sliders values to the joints angles and positions

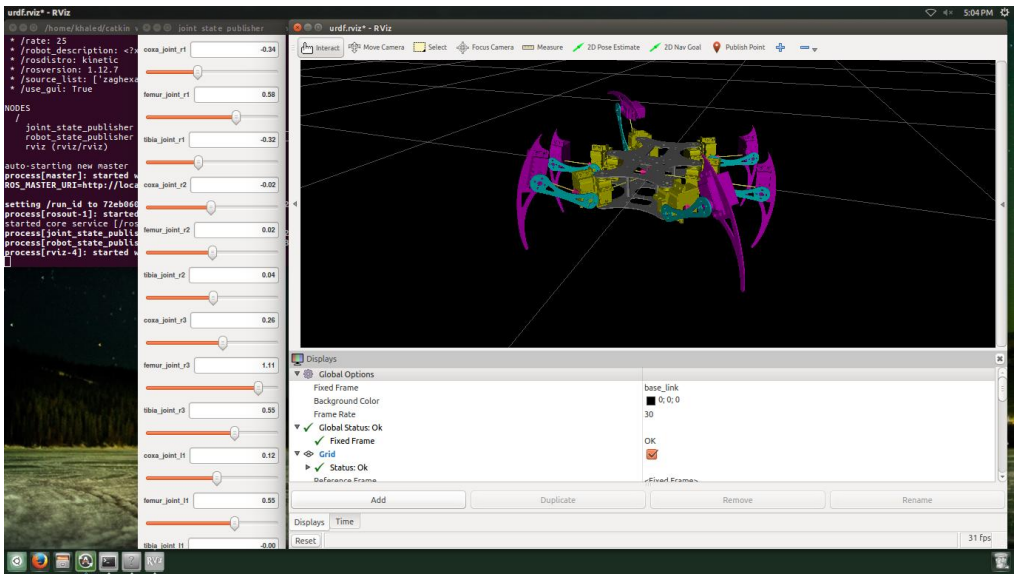

Figure 3.7.: Control sliders and their effect on the robot

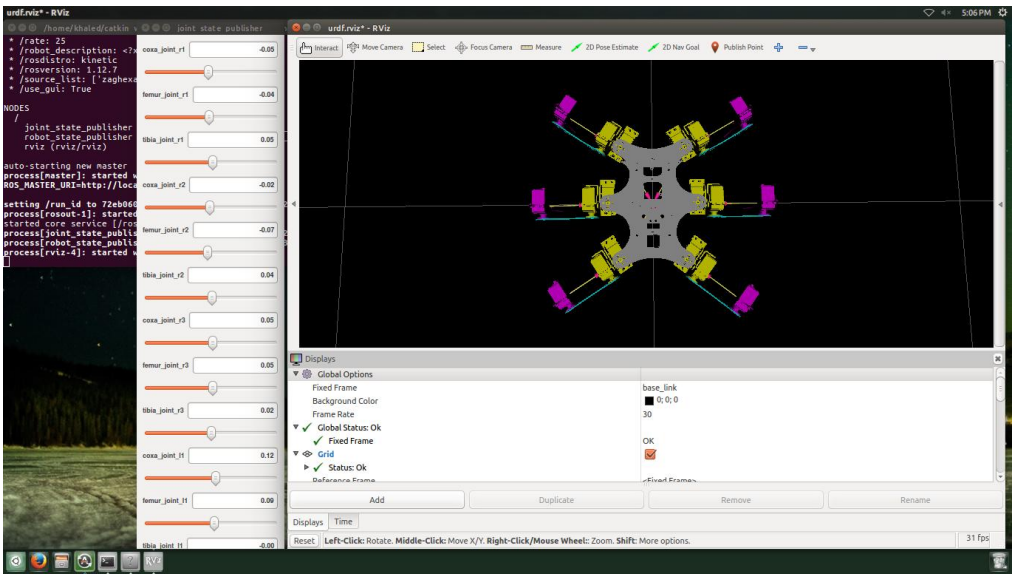

Figure 3.8.: top view of the robot

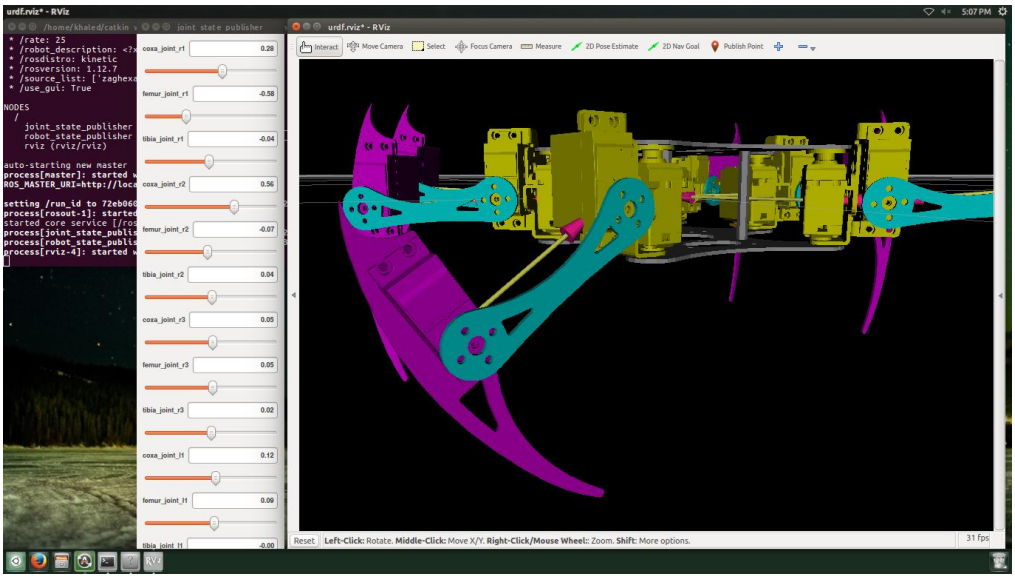

Figure 3.9.: Different views of the robot

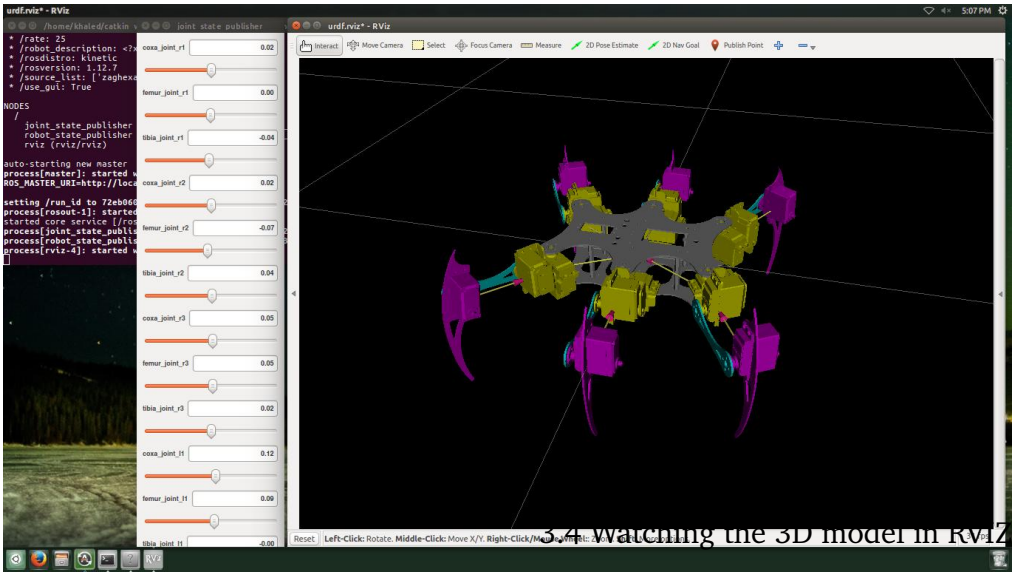

Figure 3.10.: Different views of the robot

3.5. Matlab Simulation

In the second simulation, we modelled the robot in MATLAB and employing the Robotics Toolbox. The main purpose of this simulation is to calculate and simulate the kinematics of robot. To create the six-legged walking robot, we started by creating a three-axis robot arm that we used as a leg. Then we implemented a trajectory for the leg that is suitable for walking. Finally, we instantiated six instances of the leg to create the walking robot. The equations given in Sec.4 are programmed first for one leg and tested on successful working, the whole body kinematics were also programmed and tested. The results were very useful in modifying the walking gaits of the robot which then implemented in the real robot. Figure 10 shows one such simulations in which the same experiment performed on the real robot to test the whole body kinematics for raising and lowering the body height.

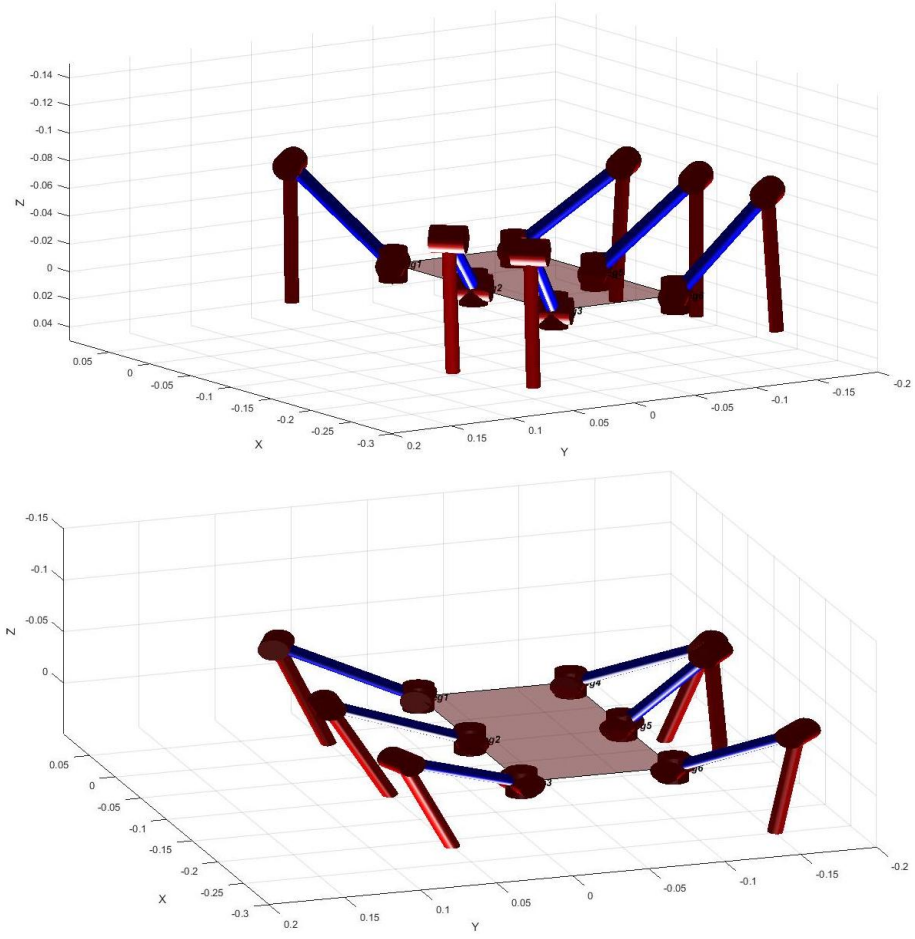

Figure 3.11.: Hexapod robot Simulation.

We will launch it with the following command:

```
$ roslaunch zaghexa_sim display_model.launch model:="`rospack find
    zaghexa_sim`/urdf/zaghexa_sim.urdf"
```

if every thing is fine and you have no errors, it will load RVIZ and you will see:

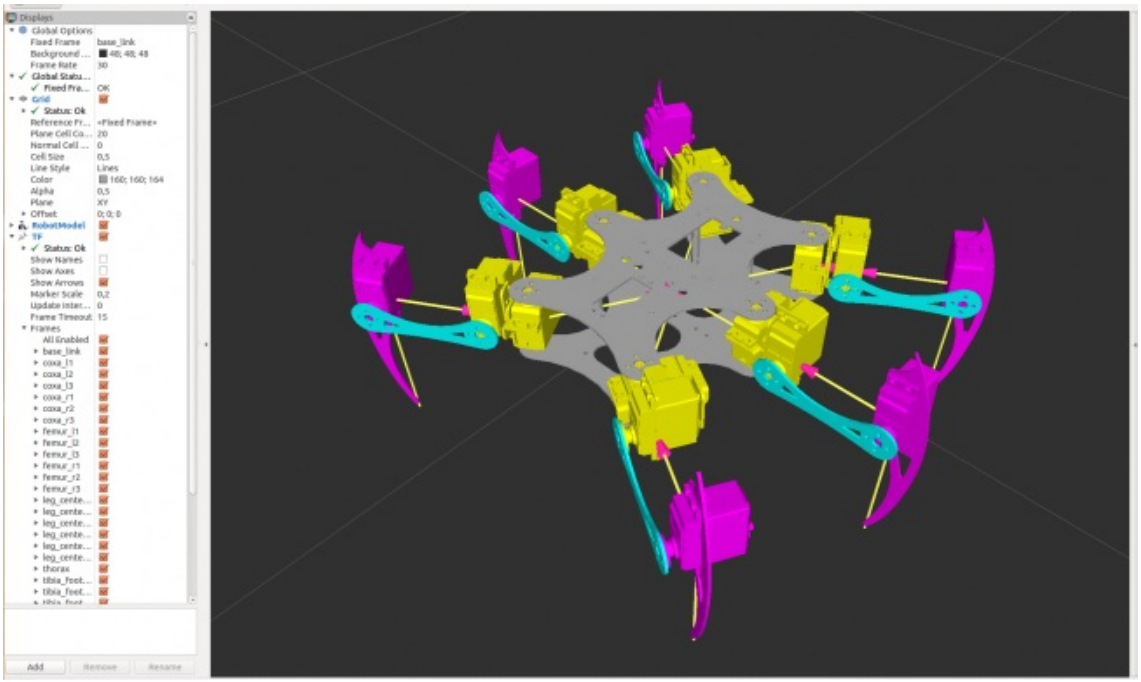

Figure 3.12.: output of urdf to graphics

3.6. Making our robot movable

A good way of testing whether or not the axis and limits of the joints are fine by running rviz with joint state publisher GUI

```
$ roslaunch zaghexa_sim display.launch model:="‘rospack find
    zaghexa_sim‘/urdf/zaghexa_sim.urdf" gui:=true
```

you will see a GUI with some sliders each of them controls one joint of the 18 joints so we have 18 sliders:
In the next figures you will see the effect of changing sliders values to the joints angles and positions

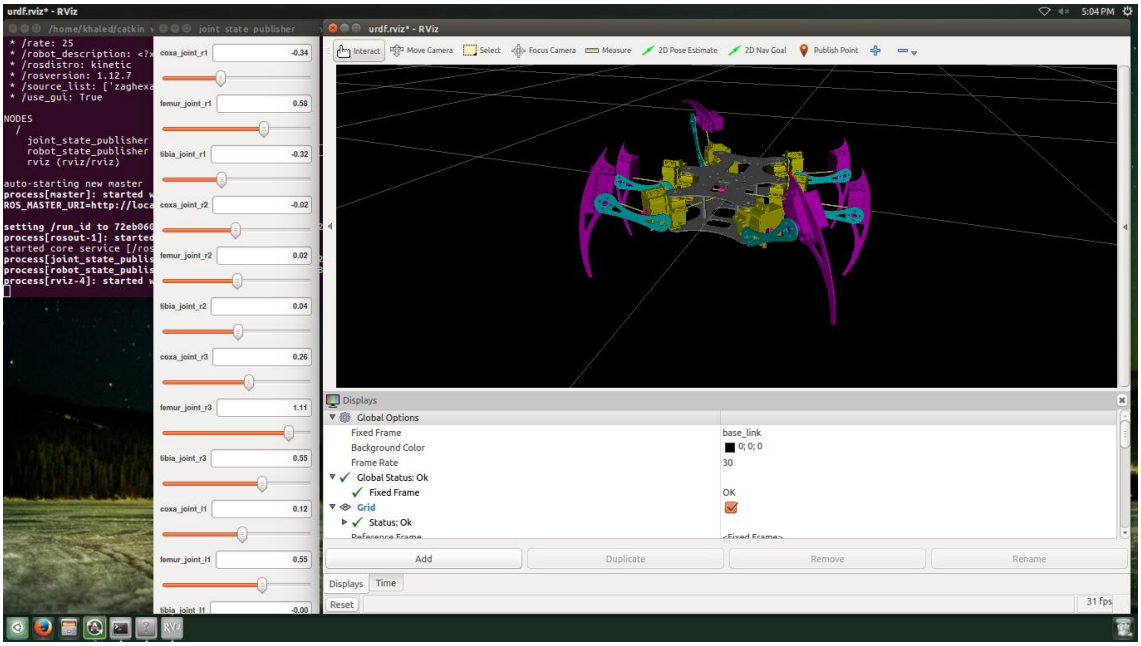

Figure 3.13.: Joint state publisher GUI with its control sliders and their effect on the robot

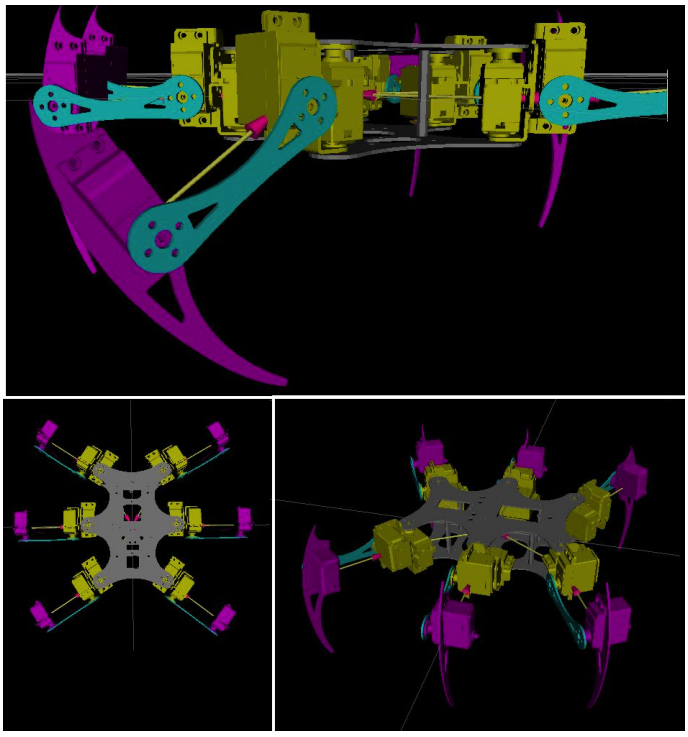

Figure 3.14.: Different views of the robot

Chapter 4
Software Architecture

The high–level functionality and control of the robot are implemented in ROS packages. The Robot Operating System (ROS) is a standard and open–source operating system for robot control [Cousins, 2011]. ROS is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communication layer above the host operating systems of a heterogeneous compute cluster. Our software ROS packages interact with each of the subsystems in C++ and Python for direct system control. The system described uses a Linux based software framework as an operating system (OS) for providing the advantages of using an OS, which supports developing additional modules that can be easily implemented and integrated.

4.1. Robot Operating System (ROS)

Robot Operating System (ROS) provides operating system like service for the robot. It is a meta-operating system, which loads on top of an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. These facilities cannot only help robots but also other embedded systems with a rich set of tools to successfully manage the complexity. With ROS handling the basic communications and data exchange.

ROS currently only runs on Unix-based platforms. Software for ROS is primarily tested on Ubuntu and Mac OS X systems, though the ROS community has been contributing support for Fedora, Gentoo, Arch Linux and other Linux platforms. While a port to Microsoft Windows for ROS is possible, it has not yet been fully explored.

The core ROS system, along with useful tools and libraries are regularly released as a ROS Distribution. This distribution is similar to a Linux distribution and provides a set

of compatible software for others to use and build upon example for this distribution: Hydro, Indigo, Jade, Kinetic, Lunar...etc. [http://wiki.ros.org]

Our choice was Kinetic because is the update distribution that have a LTS with high variety of support from ROS community that make it easy to exchange the knowledge and figure out how to solve issues and to come up with new idea to implement it.

4.1.1. Why ROS?

The answer for this question lies in power of ROS the make it preferable for most of the designers:

Inter-platform operability : ROS message-passing means that you can work between very different components and subsystems that are probably running with different languages (maybe something like low-level hardware control with C for speed, and high-level state machines with Java or Python for ease of coding). This also gets around the problem of the mess of APIs you would have had to deal with before.

Modularity : Since things are connected by a distributed message system, if one component crashes, your whole system doesn't crash. Granted, there are plenty of ways to make your system more robust so that this doesn't happen in the first place, but ROS makes it easier for your robot to continue doing its thing even if two sensors and an arm motor have died (for example).

Concurrent resource handling : Without ROS, reading/writing to resources quickly becomes a mess with large multi-threaded systems (i.e. virtually any robotics application). Again, there are ways to deal with this, but ROS simplifies the whole process by ensuring that your threads aren't actually trying to read and write to shared resources, but are rather just publishing and subscribing to messages.

Vibrant Community : Over the past several years, ROS has grown to include a large community of users worldwide. Historically, the majority of the users were in research labs, but increasingly we are seeing adoption in the commercial sector, particularly in industrial and service robotics.

Collaborative Environment : ROS by itself offers a lot of value to most robotics projects, but it also presents an opportunity to network and collaborate with the world class roboticists that are part of the ROS community. One of the core philosophies in ROS is shared development of common components.

Interested reader can find more information,material, and much more at: `www.ros.org/core-components`

4.1.2. ROS Package

Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages it to provide this useful functionality in an easy-to-consume manner so that software can be easily reused. In general, ROS packages follow a "Goldilocks" principle: enough functionality to be useful, but not too much that the package is heavyweight and difficult to use from other software.

Packages are easy to create by hand or with tools like catkin_create_pkg. A ROS package is simply a directory descended from ROS_PACKAGE_PATH (see ROS Environment Variables) that has a package.xml file in it. Packages are the most atomic unit of build and the unit of release. This means that a package is the smallest individual thing you can build in ROS and it is the way software is bundled for release (meaning, for example, there is one debian package for each ROS package), respectively.

4.1.3. ROS Node

A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one Node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provide a graphical view of the system, and so on.

The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems. Implementation details are also well hidden as the nodes expose a minimal API to the rest of the graph and alternate implementations, even in other programming languages, can easily be substituted.

All running nodes have a graph resource name that uniquely identifies them to the rest of the system. For example, /hokuyo_node could be the name of a Hokuyo driver broadcasting laser scans. Nodes also have a node type, that simplifies the process of referring to a node executable on the fileystem. These node types are package resource names with the name of the node's package and the name of the node executable file. In order to resolve a node type, ROS searches for all executables in the package with the specified name and chooses the first that it finds. As such, you need to be careful and not produce different executables with the same name in the same package. Finally, a ROS node is written with the use of a ROS client library, such as roscpp or rospy.

4.1.4. Publisher/Subscriber

The Publisher object represents a publisher on the ROS network. The object publishes to an available topic or to a topic that it creates. This topic has an associated message type. When the Publisher object publishes a message to the topic, all subscribers to the topic receive this message. The same topic can have multiple publishers and subscribers.

The primary mechanism for ROS nodes to exchange data is to send and receive messages. Messages are transmitted on a topic and each topic has a unique name in the ROS network. If a node wants to share information, it will use a publisher to send data to a topic. A node that wants to receive that information will use a subscriber to that same topic. Besides its unique name, each topic also has a message type, which determines the types of messages that are allowed to be transmitted.

This publisher/subscriber communication has the following characteristics:

1. Topics are used for many-to-many communication. Many publishers can send messages to the same topic and many subscribers can receive them.
2. Publisher and subscribers are decoupled through topics and can be created and destroyed in any order. A message can be published to a topic even if there are no active subscribers.

The concept of topics, publishers, and subscribers is illustrated in the figure.**??**.

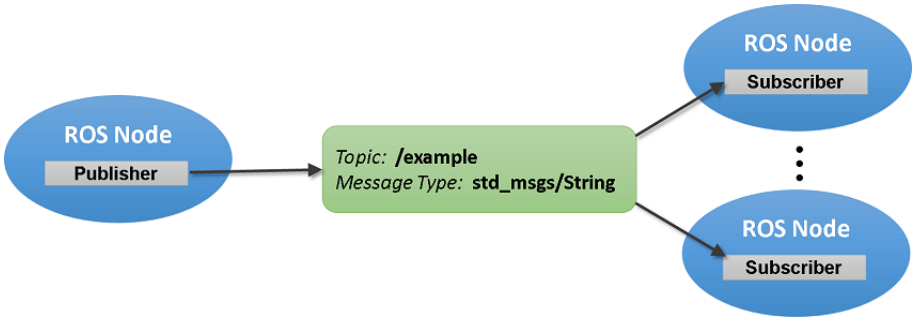

Figure 4.1.: publish subscribe concept

4.1.5. ROS Topic

Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.

Topics are intended for unidirectional, streaming communication. Nodes that need to perform remote procedure calls, i.e. receive a response to a request, should use services instead. There is also the Parameter Server for maintaining small amounts of state.

Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching type. The Master does not enforce type consistency among the publishers, but subscribers will not establish message transport unless the types match. Furthermore, all ROS clients check to make sure that an MD5 computed from the msg files match. This check ensures that the ROS Nodes were compiled from consistent code bases.

with a simple words we can define this terms:

Nodes: A node is an executable that uses ROS to communicate with other nodes. Topics: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages. Messages: ROS data type used when subscribing or publishing to a topic. Master: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services. roscore: is a collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate. It is launched using the roscore command.

4.1.6. Can we access ROS form a remote computer?

ROS is a distributed computing environment. A running ROS system can comprise dozens, even hundreds of nodes, spread across multiple machines. Depending on how the system is configured, any node may need to communicate with any other node, at any time.

- There must be complete, bi-directional connectivity between all pairs of machines, on all ports.
- Each machine must advertise itself by a name that all other machines can resolve.

More information can be found at: `wiki.ros.org/ROS/NetworkSetup`

4.2. UNIX

UNIX is an open source operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops. UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use

environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available.

- Types of UNIX

There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, Ubuntu, Kali, and MacOS X. Here in the School, we use Solaris on our servers and workstations, and Fedora Linux on the servers and desktop PCs.

- The UNIX operating system The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

- The kernel The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls.

- The shell The shell acts as an interface between the user and the kernel. The shell is a command line interpreter (CLI), it interprets the commands the user types in and arranges for them to be carried out.

4.3. General View Of The Control System

The control system of the robot can be divided into two parts:

- Raspberry Pi - High level -
- Arduino - Low level -

All computing and control operation is done on RPi and Arduino that lies on the robot to inure the speed of control and precision. The sensors data send from mobile App to the RPi through TCP/IP connection and to a computer also, RPi receive this data and process it to be aware with its environment and make a decision upon it, the computer show this data on a GUI to the operator/user so he can also make a decision using the reading form sensors and the camera. When a order form an operator using Joystick/Bluetooth to move the robot the RPi take this order and send it to the Arduino through a serial port, using a C++ program in Arduino we compute the angles to move to the desired direction with minimal error. FigureFigure 4.2 show an Outline for the control system.

4.3.1. Raspberry Pi

A Raspberry Pi is a credit card-sized computer originally designed for education, inspired by the 1981 BBC Micro. Creator Eben Upton's goal was to create a low-cost device that would improve programming skills and hardware understanding. But thanks to

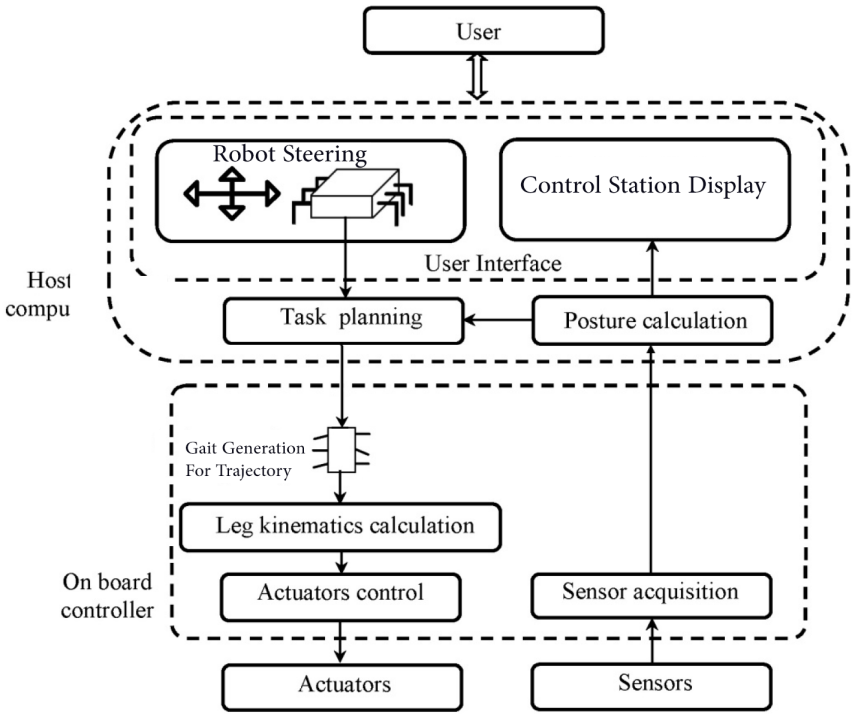

Figure 4.2.: Outline of control systems.

its small size and accessible price, it was quickly adopted by tinkerers, makers, and electronics enthusiasts for projects that require more than a basic microcontroller.

The Raspberry Pi is slower than a modern laptop or desktop but is still a complete Linux computer and can provide all the expected abilities that implies, at a low-power consumption level.

The Raspberry Pi is open hardware, with the exception of the primary chip on the Raspberry Pi, the Broadcomm SoC (System on a Chip), which runs many of the main components of the board–CPU, graphics, memory, the USB controller, etc. Many of the projects made with a Raspberry Pi are open and well-documented as well and are things you can build and modify yourself.

- What kind of operating system does the Raspberry Pi run?

  The Raspberry Pi was designed for the Linux operating system, and many Linux distributions now have a version optimized for the Raspberry Pi.

  Two of the most popular options are Raspbian, which is based on the Debian operating system, and Pidora, which is based on the Fedora operating system. For beginners, either of these two work well; which one you choose to use is a matter of personal preference. A good practice might be to go with the one which most closely resembles an operating system you're familiar with, in either a desktop or server environment.

If you would like to experiment with multiple Linux distributions and aren't sure which one you want, or you just want an easier experience in case something goes wrong, try NOOBS, which stands for New Out Of Box Software. When you first boot from the SD card, you will be given a menu with multiple distributions (including Raspbian and Pidora) to choose from. If you decide to try a different one, or if something goes wrong with your system, you simply hold the Shift key at boot to return to this menu and start over.

There are, of course, lots of other choices. OpenELEC and RaspBMC are both operating system distributions based on Linux that are targeted towards using the Raspberry Pi as a media center. There are also non-Linux systems, like RISC OS, which run on the Pi. Some enthusiasts have even used the Raspberry Pi to learn about operating systems by designing their own.

Raspberry Pi 3 Model B specification The Raspberry Pi 3 Model B is the third generation Raspberry Pi. This powerful credit-card sized single board computer can be used for many applications and supersedes the original Raspberry Pi Model B+ and Raspberry Pi 2 Model B. Whilst maintaining the popular board format the Raspberry Pi 3 Model B brings you a more powerful processer, 10x faster than the first generation Raspberry Pi. Additionally it adds wireless LAN & Bluetooth connectivity making it the ideal solution for powerful connected designs.

Processor	Broadcom BCM2387 chipset. 1.2GHz Quad-Core ARM Cortex-A53 802.11 b/g/n Wireless LAN and Bluetooth 4.1 (Bluetooth Classic and LE)
GPU	Dual Core VideoCore IV® Multimedia Co-Processor. Provides Open GL ES 2.0, hardware-accelerated OpenVG, and 1080p30 H.264 high-profile decode.
Memory	1GB LPDDR2
Operating System	Boots from Micro SD card, running a version of the Linux operating system
Dimensions	85 x 56 x 17mm
Power	Micro USB socket 5V1, 2.5A
Ethernet	10/100 BaseT Ethernet socket
Video Output	HDMI (rev 1.3 & 1.4 Composite RCA (PAL and NTSC)
Audio Output	Audio Output 3.5mm jack, HDMI
USB	4 x USB 2.0 Connector
GPIO Connector	40-pin 2.54 mm (100 mil) expansion header: 2x20 strip Providing 27 GPIO pins as well as +3.3 V, +5 V and GND supply lines
Camera Connector	15-pin MIPI Camera Serial Interface (CSI-2)
Display Connector	Display Serial Interface (DSI) 15 way flat flex cable connector with two data lanes and a clock lane
Memory Card	Slot Push/pull Micro SDIO

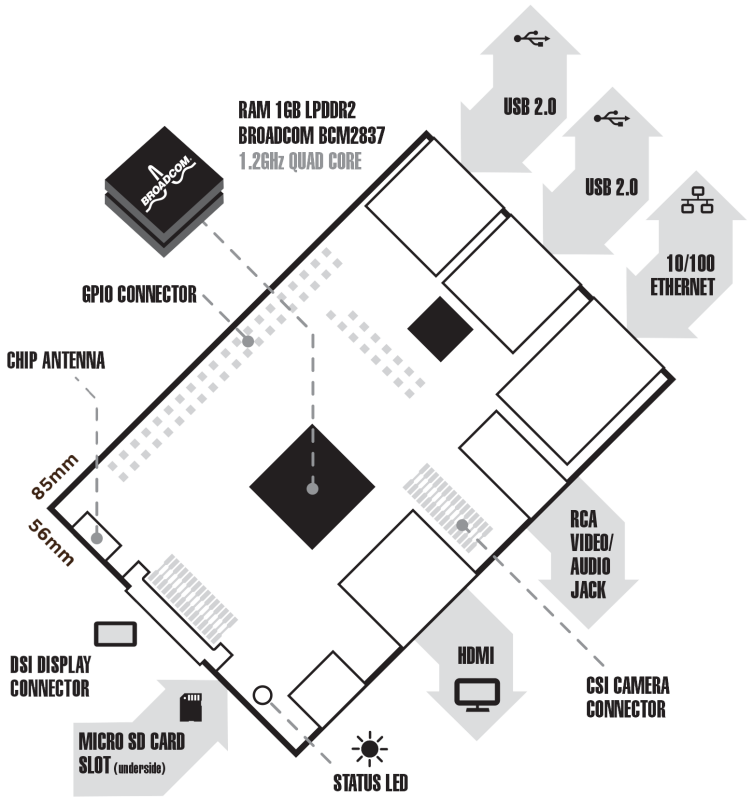

Figure 4.3.: Raspberry Pi 3 Model B.

4.3.1.1. Ubuntu MATE with Raspberry Pi

Ubuntu MATE is a stable, easy-to-use operating system with a configurable desktop environment. It is ideal for those who want the most out of their computers and prefer a traditional desktop metaphor. With modest hardware requirements it is suitable for modern workstations, single board computers and older hardware alike. Ubuntu MATE makes modern computers fast and old computers usable.

The MATE desktop is essentially a continuation of the traditional Gnome 2 desktop environment. Combining the MATE desktop with Ubuntu gives you an experience that is almost identical to the early days of Ubuntu, prior to the switch to Unity.

Martin Wimpress and Rohith Madhavan have made an Ubuntu MATE image for the Raspberry Pi 2 and Raspberry Pi 3 based on the regular Ubuntu armhf base, not the new Ubuntu “Snappy” Core, which means that the installation procedure for applications uses the traditional tools, ie apt-get with highly recommended microSDHC Class 6 or Class 10 microSDHC card. Ubuntu MATE 16.04 also fully supports the built-in Bluetooth and Wifi on the Raspberry Pi 3 and features hardware accelerated video playback in VLC and hardware accelerated decoding and encoding in ffmpeg

To installing Ubuntu MATE onto Raspberry Pi 3, starting by download the OS into

microSD then the partition is automatically resized to fill your microSD card when the pi is powered up for the first time, typical guided installer is shown up. Installation takes several minutes and finally the system reboots and you arrive at the desktop. A Welcome app provides some good information on Ubuntu MATE, including a section specific for the Raspberry Pi.

The Welcome app explains that the while the system is based on Ubuntu MATE and uses Ubuntu armhf base, it is in fact using the same kernel as Raspian. It also turns out that a whole set of Raspian software has been ported over such as raspi-config, rpi.gpio, sonic-pi, python-sent-hat, omxplayer, etc.

[ubuntu-mate.org/blog/ubuntu-mate-xenial-raspberry-pi]

4.3.1.2. Raspberry Pi with ROS

This part is consider as the the brain of the robot as it contain all the nodes of our robot that make it move and interact with its environment, we can divide our program to two main packages that will control all functions in our robot:

1. Movement package
   This package is responsible to handle the movement of the robot,
2. Client package
   This package is responsible to handle the communication through the TCP/IP connection,

4.3.2. Arduino Mega

Arduino is an open-source platform used for building electronics projects. Arduino consists of both a physical programmable circuit board (often referred to as a microcontroller) and a piece of software, or IDE (Integrated Development Environment) that runs on your computer, used to write and upload computer code to the physical board.

The Arduino platform has become quite popular with people just starting out with electronics, and for good reason. Unlike most previous programmable circuit boards, the Arduino does not need a separate piece of hardware (called a programmer) in order to load new code onto the board – you can simply use a USB cable. Additionally, the Arduino IDE uses a simplified version of C++, making it easier to program. Finally, Arduino provides a standard form factor that breaks out the functions of the microcontroller into a more accessible package.

The Arduino Mega 2560 is a microcontroller board based on the ATmega2560. It has 54 of digital input/output pins (14 can be used as PWM outputs), 16 analog inputs, a USB connection, a power jack,a 16 MHz crystal oscillator, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started. The

large number of pins make this board very handy for projects that require a bunch of digital inputs or outputs.

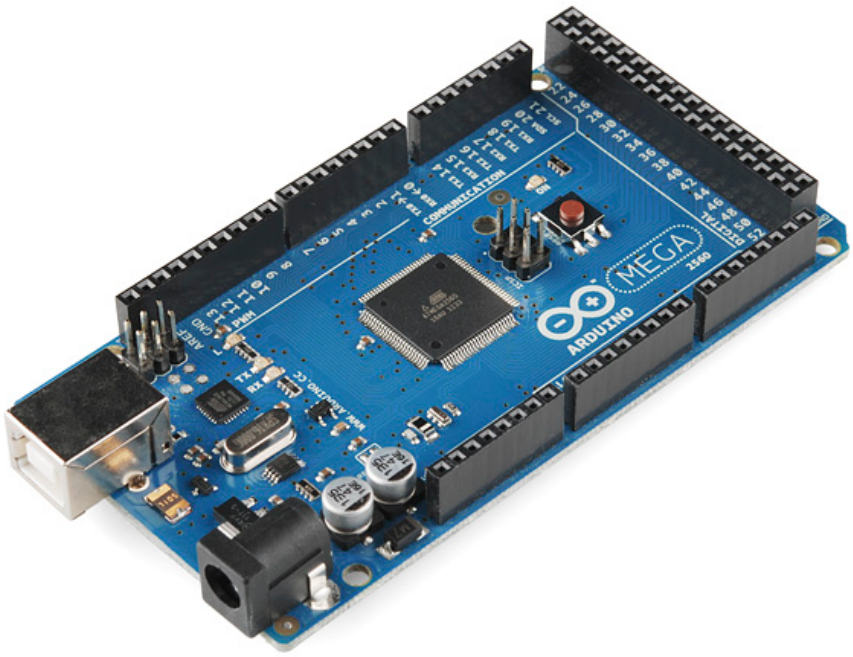

Figure 4.4.: Arduino Mega.

Arduino Mega specification

Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage	(recommended) 7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 14 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz

4.3.2.1. Arduino whit Raspberry Pi

Rather than struggle with the very basic unprotected IO pins on the Raspberry Pi and the lack of real-time performance in Linux, the ideal setup for many real-world-interfacing

projects is Raspberry Pi + Arduino.

There are four basic ways to connect Arduino to Raspberry Pi:

1. Buy an add-on board like the Gertboard which has an Arduino compatible IC on it. Pricey.
2. Plug a standard Arduino into the USB port of the RPi. This is by far the easiest method and minimises wiring and hassle. However it requires the more expensive Arduinos.
3. Use a USB to Serial adapter with a cheaper/smaller Arduino like a Pro Mini or a self-made Shrimp. This is the best DIY option and has the same advantage of method 2 that you can power the Arduino/Shrimp from USB.
4. Use the Serial Pins on the Raspberry Pi to connect to a cheaper/smaller Aruduino like a Pro Mini or a self-made Shrimp. This is theoretically the cheapest method but by far the most hassle. This is also the best method if you are using the cheaper Raspberry Pi Model A and its single USB port is being used for Wifi.

Using the second method to connect the two devices in serial connection,

4.3.2.2. Arduino PWM to Servo Motors

4.4. Mobile Application

Most of the android devices have built-in sensors that measure motion, orientation, and various environmental condition. SensorsCORE is Android application act like server that read any kind of data from your smartphone's sensors and send it through socket request to any client.

- How to communicate:

The communication between SensorsCOER(server) and your application(client) based on socket communication. The server is accessed via server's IP address (or domain name) and a port number. Once two application are connected, they can communicate streams of bytes with each other.

- How it works:

Once your application connected with the server for the first time, it send a list of your available sensors in your smartphone - you can use – with its IDs. You can send ID of the desired sensor that you want to get its values to the server. Every time you send a request, you get an answer with the sensor values. So we have three type of requests. The first request sent with first time connection with the list of available sensors and its IDs. When you want a data form specific sensor, you sent a request with the sensor's id The third request is initialized when you want the data from all sensors.

Example:

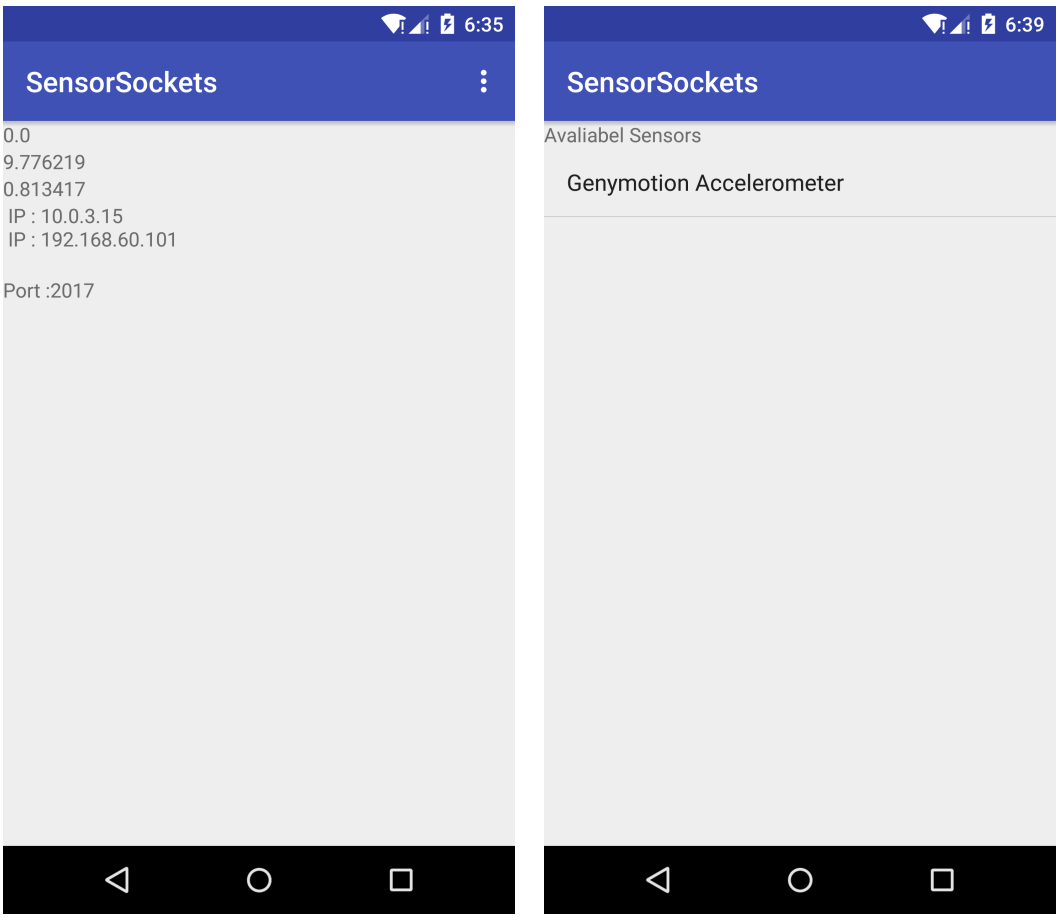

Figure 4.5.: Home page (left) and Sensor window (right)

4.5. Communication Types

4.5.1. Joystick

the wireless joystick communicates with the Raspberry Pi, using a ROS node called 'joy', and send a keystroke to it. Then the Raspberry Pi find the key pressed using a python library called 'pygame' and send a proper character on the serial port to the Arduino, our low-level interface with the servomotors. Lastly, the Arduino checks the received character and perform the needed action.

4.5.2. Bluetooth

4.5.3. Wi-Fi

Another line of communication is between our control station and an Android smart phone mounted on the robot through a TCP/IP client server application.

Figure 4.6.: JoyStick.

- What is TCP/IP?
  When two computers follow the same protocols—the same set of rules—they can understand each other and exchange data. TCP/IP (Transmission Control Protocol/Internet Protocol) is the basic communication language or protocol of the Internet that use this concept to communicate through the internet and become standard terminology for referring to this suite of protocols. TCP/IP architecture omits some features found under the OSI model, combines the features of some adjacent OSI layers and splits other layers apart. The 4-layer structure of TCP/IP is built as information is passed down from applications to the physical network layer. When data is sent, each layer treats all of the information it receives from the upper layer as data, adds control information (header) to the front of that data and then pass it to the lower layer. When data is received, the opposite procedure takes place as each layer processes and removes its header before passing the data to the upper layer.

  - Application Layer
  The Application Layer in TCP/IP groups the functions of OSI Application, Presentation Layer and Session Layer. Therefore any process above the transport layer is called an Application in the TCP/IP architecture. In TCP/IP socket and port are used to describe the path over which applications communicate. Most application level protocols are associated with one or more port number. Transport Layer. In TCP/IP architecture, there are two Transport Layer protocols. The Transmission Control Protocol (TCP) guarantees information transmission. The User Datagram Protocol (UDP) transports datagram without end-to-end reliability checking. Both protocols are useful for different applications.

  - Network Layer
  The Internet Protocol (IP) is the primary protocol in the TCP/IP Network Layer. All upper and lower layer communications must travel through IP as they are passed through the TCP/IP protocol stack. In addition, there are many supporting protocols in the Network Layer, such as ICMP, to facilitate and manage the routing process.

  - Network Access Layer

In the TCP/IP architecture, the Data Link Layer and Physical Layer are normally grouped together to become the Network Access layer. TCP/IP makes use of existing Data Link and Physical Layer tandards rather than defining its own. Many RFCs describe how IP utilizes and interfaces with the existing data link protocols such as Ethernet, Token Ring, FDDI, HSSI, and ATM. The physical layer, which defines the hardware communication properties, is not often directly interfaced with the TCP/IP protocols in the network layer and above. []

TCP/IP uses the client/server model of communication in which a computer user (a client) requests and is provided a service (such as sending a Web page) by another computer (a server) in the network. TCP/IP communication is primarily point-to-point, meaning each communication is from one point (or host computer) in the network to another point or host computer. TCP/IP and the higher-level applications that use it are collectively said to be "stateless" because each client request is considered a new request unrelated to any previous one (unlike ordinary phone conversations that require a dedicated connection for the call duration). Being stateless frees network paths so that everyone can use them continuously. (Note that the TCP layer itself is not stateless as far as any one message is concerned. Its connection remains in place until all packets in a message have been received.)

- Protocol A protocol is the special set of rules that end points in a telecommunication connection use when they communicate. Protocols specify interactions between the communicating entities. Protocols exist at several levels in a telecommunication connection. For example, there are protocols for the data interchange at the hardware device level and protocols for data interchange at the application program level. In the standard model known as Open Systems Interconnection (OSI), there are one or more protocols at each layer in the telecommunication exchange that both ends of the exchange must recognize and observe. Protocols are often described in an industry or international standard.

  The TCP/IP Internet protocols, a common example, consist of: Transmission Control Protocol (TCP), which uses a set of rules to exchange messages with other Internet points at the information packet level Internet Protocol (IP), which uses a set of rules to send and receive messages at the Internet address level Additional protocols that include the Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP), each with defined sets of rules to use with corresponding programs elsewhere on the Internet

- The Protocol Used In ZagHexa Robot

  Client will send "I" character in the first time of connection, Server will reply by an array of string in this format "Sensor1_name, ID, Sensor2_name,ID, ..etc" Ex: "gyroscope, 3, GPS, 4, ..etc"

  Client will store this array into an text file to retrieve the ID for future requests. Client can do one of these requests :

- "A" character to send all sensors data.
- ID number to select one sensor to get its data.

Server will reply in the first case by an array of numbers in the same order of the sensors in the index file, in the second case it will reply by a single float number that its index match the request one.

4.6. GUI

Future work

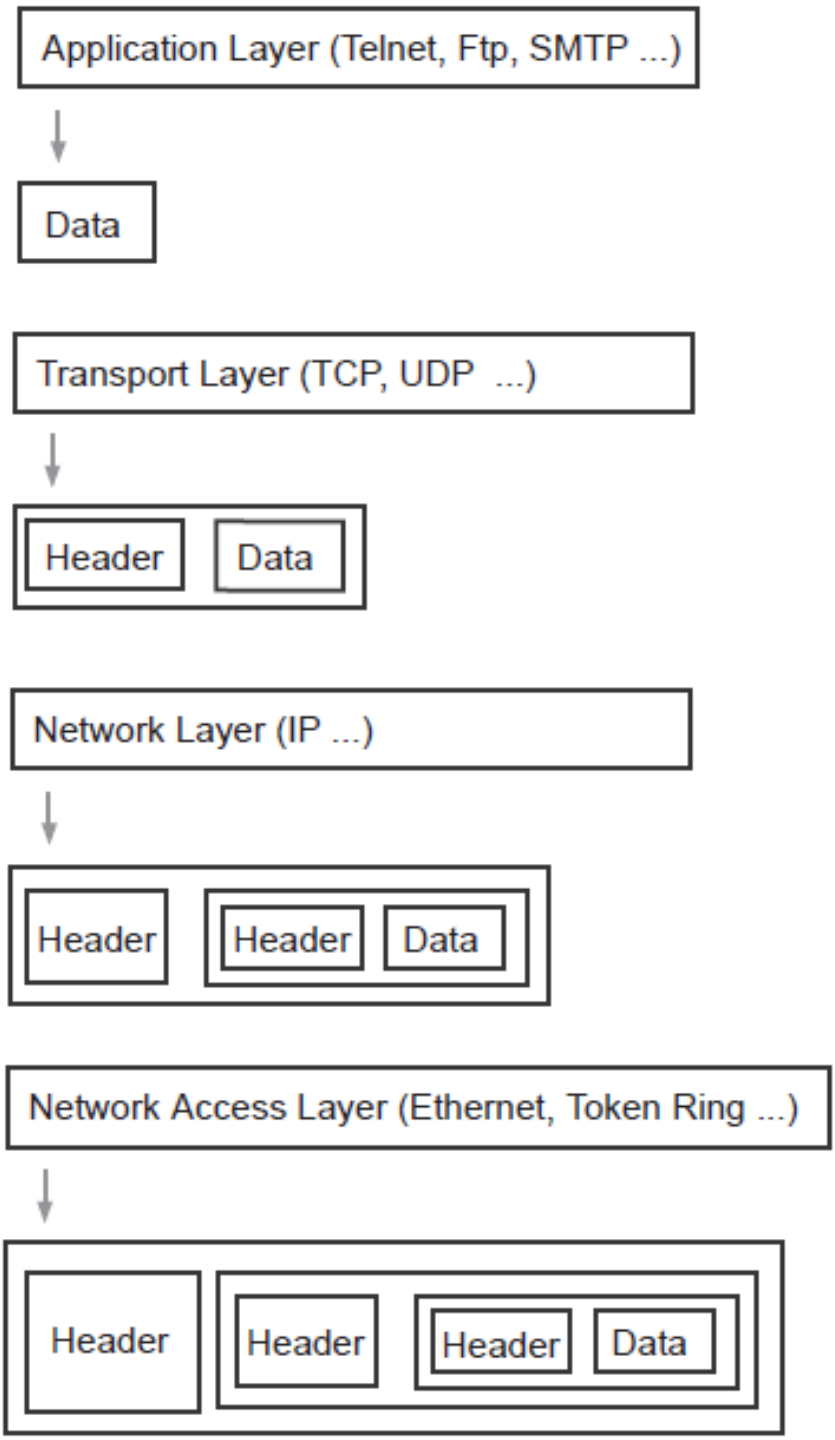

Figure 4.7.: TCP/IP Protocol.

"A computer would deserve to be called intelligent if it could deceive a human into believing that it was human."

— Alan Turing, (British pioneering computer scientist, cryptanalyst,···, and philosopher, 1912–1954)

Chapter 5
Mechanical Design

5.1. Design Characteristics

In the following section, the main design characteristics and modeling issues are discussed in order to show how the proposed design procedure can be implemented. It is worth mentioning also that other design characteristics exist and additional features can be considered for specific applications. Thus, the most remarkable design characteristics have been considered as examples for filling in the data in the cause and effect matrix.

5.1.1. Robot Body Architecture

There are two basic architectures of hexapod robots: rectangular and hexagonal as shown in Figure 5.1. The first one has six legs distributed symmetrically along two sides,each side having three legs. The second haslegs distributed axi-symmetrically around the body, in a hexagonal or circular shape.

Many references can be found in the literature on rectangular six-legged robots, they describe the longitudinal stability margin for rectangular hexapods. Also, the feasible walking gaits have been widely investigated and tested. Bilateral symmetry may be better suited than radial symmetry to move along a straight line. Rectangular architectures require a special gait for turning action; generally, they need four steps in order to realize a turning action. Hexagonal hexapod robots demonstrate better performances than rectangular robots for some aspects. As example hexagonal robots can have many kinds of gaits and can easily change direction—in fact true radial symmetry implies that all legs are equal and the body has no "front" or "rear"—there is thus no preferential directionfor the motion. It is proved that hexagonal hexapods can easily steerin all directions and that they have a longer stability margin. It isfound that hexagonal robots rotate and move in all directions at the same time,better than

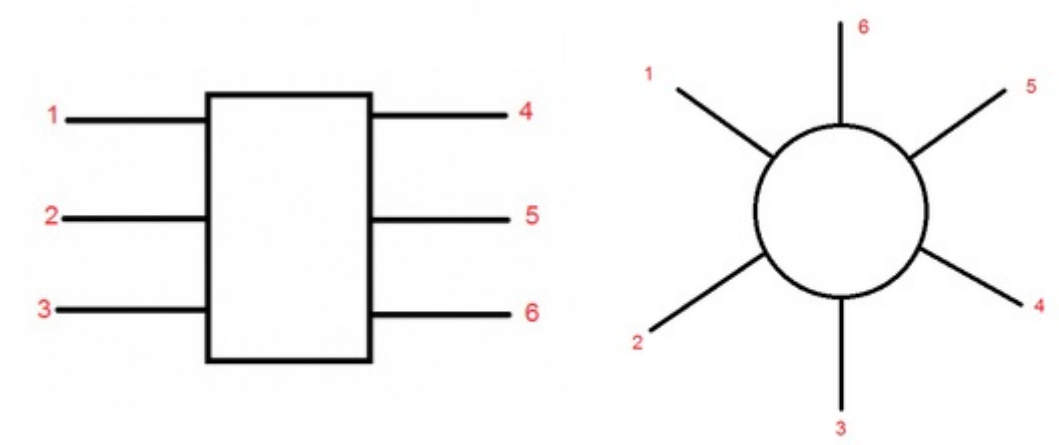

Figure 5.1.: Basic architectures of hexapod robots

rectangular ones, by comparing stability margin and stroke in wave gait.It is proved theoretically that hexagonal hexapod robots have superior stability margin,stride and turning ability compared to rectangular robots.

5.1.2. Kinematic Architectures of Legs

Kinematics architecture depends on the factors related to the application in which the hexapod robot is required for, as for example the terrain form, the workspace and the payload. Literature shows that there is a number of different leg types currently employed for hexapod walking robots. All have advantages and disadvantages. Figure 5.2 shows a schematic classification of hexapod legs types.

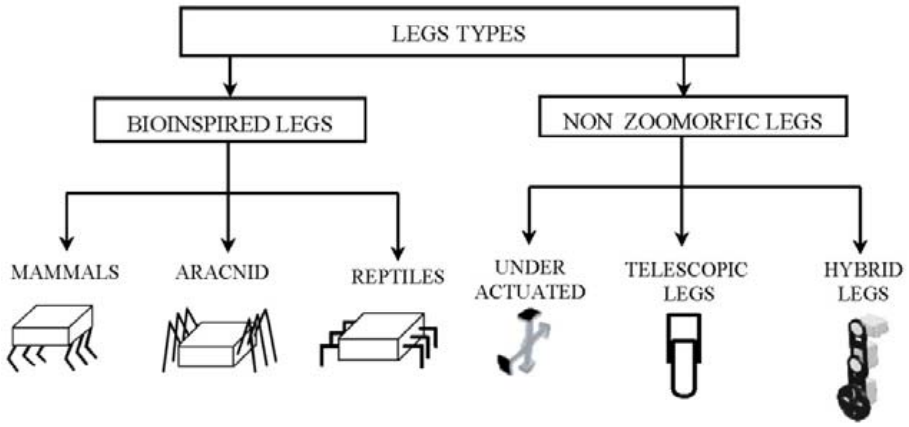

Figure 5.2.: Classification of hexapod legs types

At the first stage, one can choose between bio inspired and non-zoomorphic legs. Bio inspired leg configuration is motivated primarily by animal gait, such as reptiles,

mammals or arachnid. The first one has legs and bodies for moving over rough and uneven terrains. The principal characteristic of the Reptilian type is that the legs are placed on both ends of the protruding body and knees to the side of the base. Mammals' bodies are above the legs, which gives less support to the base and lower power consumption is needed to support the body, but it requires more stability than other types of animal. In Arachnid configuration, legs extremities are situated on both sides, sticking the knees at the top of the spider's body. The orientation of the legs in respect to the body of the hexapod robot can be done with three configurations: frontal, sagittal or circular as shown in Figure 5.3 .

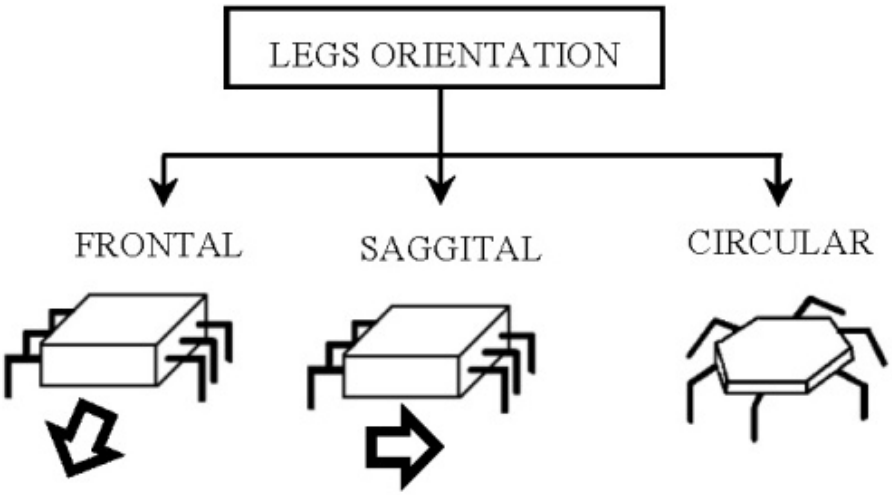

Figure 5.3.: Hexapod leg configurations

In the first one, the directions are perpendicular to the advancement of the legs' position, unlike the sagittal, which moves parallel to the robot legs, while in the circular arrangement the legs are positioned radially to the body of the system allowing the mechanism to move in any direction. In the mammalian configuration, the legs are below the body and can place the knees in different positions depending on the application it requires, shown in figure (d). Non zoomorphic legs can be hybrids, telescopic or under-actuated, a solution named Roller-Walker is presented. The principle through which the robot propels itself during wheeled locomotion is the same as that of the skaters.

5.1.3. Actuator Types

Many kinds of actuators can be employed for operating hexapod walking robots. The majority of hexapods is actuated by electric rotating motors (Servo motors), as they are relatively cheap, easy to control and there are suitable technologies to store the energy.

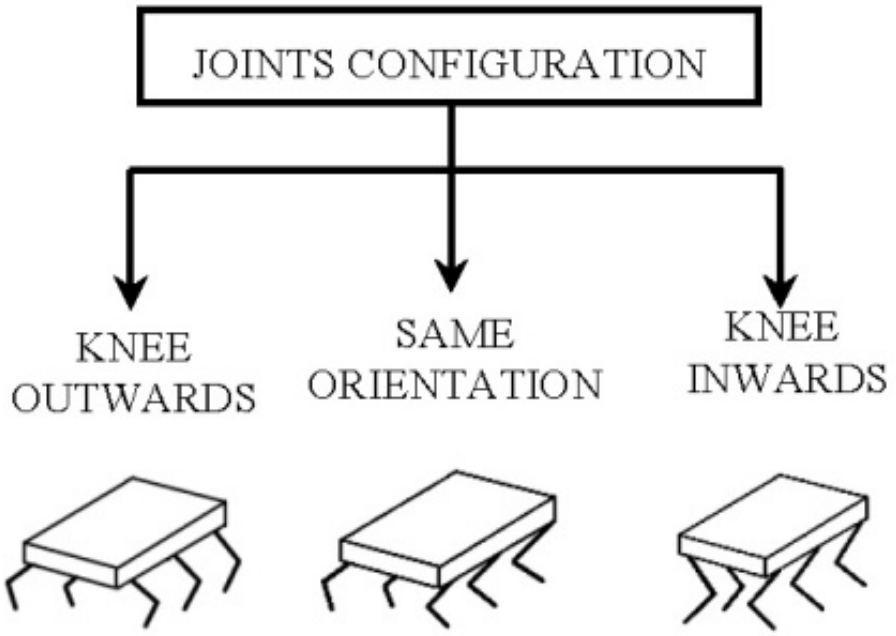

Figure 5.4.: Joint configurations

Linear motors are able to generate considerable forces at very considerable speeds. However, these do not yet appear to have been utilized in many hexapods since they have a limited movable range to weight ratio. Pneumatics actuators have very low stiffness, inaccurate response, and low power to weight ratio. Pneumatics actuators, or air muscle, are able to offer a fast response time but they need an on-board air supply as bottles or compressors that are heavy pieces of equipment. Hydraulics actuators have high power/weight ratio; they are able to supply very high force, but suffer from the serious drawback of having to carry an engine to drive the pump. Hydraulics actuators are suitable for larger sized hexapod robots.

Unconventional actuators for hexapod robots can also be materials that can change shape through the direct application of electricity or a chemical agent. Ionic polymer-metal composites, for example, are materials that exhibit high strains under applied voltage differences allowing them to change from a sheet shape into curved shapes. Polyacrylonitrile is a form of artificial silk, classed as a gel polymer. It can respond fast, but the activation method is a change in pH, which requires the fibers to be housed in a watertight bag. Another class of materials that can change shape under the application of electricity is the Shape Memory Alloys (SMA), such as a nickel-titanium alloy that exhibits extreme contraction when heated.

Contraction rates are controlled by the heating and cooling; the major drawback resulting in slow response times and small operating forces. Thus, despite a great research interest and the building of some prototypes, the uses of SMA as hexapod actuators have been very limited. Present trends indicate that they are more suited to micro robots.

5.1.4. Actuators Arrangements

Typically, specific actuator arrangements are developed in order to obtain maximum leg workspace with a minimal kinematic structure. Several types of geometrical arrangements such as in Figure 5.5 are recurrent in literature.

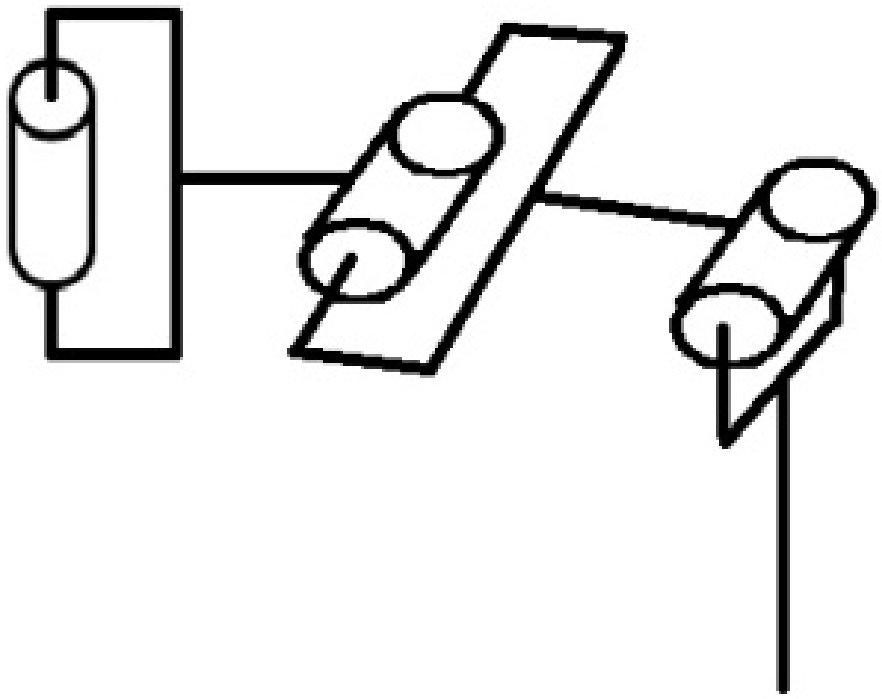

Figure 5.5.: Three DoF solution

The design consists of links connected through knee joints. The walking motion is accomplished by controlling the angle of the links to position the feet. There are a number of different ways in which the joints can be actuated. Options include mounting the motor at the joint itself, or using a pulley and belt Figure 5.6 or lead screw **??** to set the angle of the knee using an actuator mounted near the base of the leg.

The major drawback of last design is the necessity to actuate remote joints. On the other hand, latching the actuator at the knee joint adds various dynamic effects to the leg which have to be compensated by the controller. This adds complexity to the control algorithms needed to move the leg.
It also requires more powerful motors at the hip joint to move the added mass of the leg. Remote actuation, in which the actuators are located at the base of the leg, eliminates some of these problems, at the cost of increasing the complexity of the mechanism. The coupling of the motion of the end effector relative to the actuators is another undesirable characteristic of this leg design. Another potential leg design is modeled according to a typical mammalian leg with a four-bar linkage structure. The major drawback of this design is that the motions are highly coupled and the effective

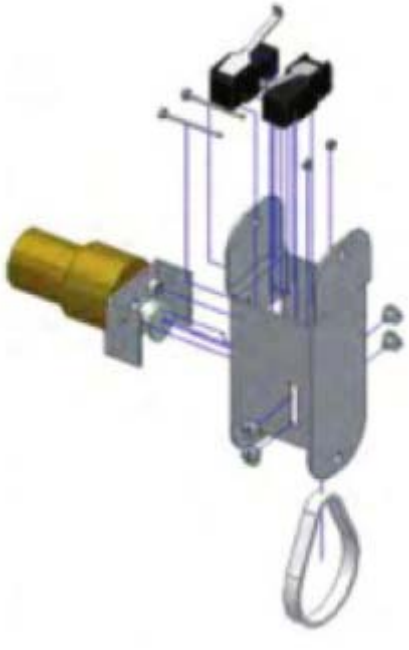

Figure 5.6.: Pinion-belt arrangement

workspace is somewhat limited. Moreover, the entire weight of the robot is supported by the hip joint and they necessitate a powerful and expensive motor.

5.2. Walking Gaits

A gait is a sequence of leg motions coordinated with a sequence of body motions for moving the overall body of the robot in the desired direction and for orientation from one place to another. A gait is described as periodic when similar states of the same leg during successive strokes occur at the same interval for all legs. Periodic gaits are suitable for smooth terrain and they have been studied by several investigators. Figure 9 shows the scheme of some periodical hexapod gaits; white color indicates that the foot is in ground contact and the black color otherwise. Figure 9a reports the hexapod legs' description.

5.2.1. Tripod gait

Tripod gait is a regular, periodic gait where the anterior and posterior legs on one side lift in time with the contralateral middle leg, forming alternating tripods. Thus, it is based on two groups of legs. During each step the first group of the legs is lifted and is rotated forward and is laid upon on the ground. Then the other group is lifted. Now both groups are moving, the first group backward, the second group forward and

Figure 5.7.: Lead screw leg

finally the second group is laid on the ground. It is obvious that both groups perform the same movement, but they are shifted by half a period. Tripod gait is very fast, but also very unstable. That is because at one moment half of the whole weight of the robot is only on one leg, which can lead to slip or even to fall. This is a gait suitable for high speed walking over relatively flat ground.

5.2.2. Wave Gait

Another gait is wave, which is the most stable gait, but also the slowest. Wave gait consists of a sequential adjustment of legs forward and only when all the legs are set to the new positions, the step is completed. In each phase of step maximally one leg is lifted up, which leads to high stability of this gait.

5.2.3. Ripple Gait

Ripple gait is inspired by insects. Each leg performs the same move – up, forward, down, backward.

Figure 5.8 shows the movement of each leg in time. A high value represents leg movement and low values means no movement. Tripod, wave and ripple gaits are shown in this figure. Tripod has two group of legs, all the legs in the same group move

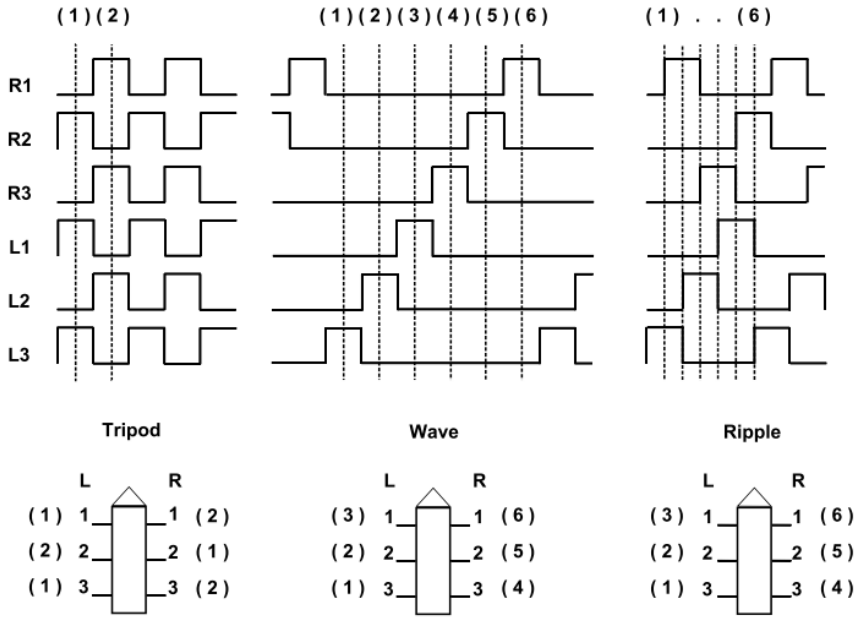

Figure 5.8.: Gait diagram of hexapodal gaits

at once. In the wave gait only one leg is moving at any time. After all legs are set up to their new positions, step is completed. In the ripple gait all legs move the same way, but their moves are shifted. Leg moves partially overlap. In other words, the time when the first foot is lifted and begins to move forward, the second leg begins to lift up. In this way the robot cycles through all legs.

These are the most common gaits. Theoretical number of different gaits N can be calculated using equation: N = (2K - 1)!, where K is the number of the legs of the robot. Not all of them are usable for effective locomotion. For hexapod robot it is 11! = 39 916 800 possibilities of locomotion. The number is quite large, because this equation calculates all possible motions, like motion up and down, which of course doesn't lead to an effective movement.

5.3. Robot Kinematics

5.3.1. Robot body frame

The origin of the robot base frame will be in the center of the body, structured with Z-axis pointing up, the X-axis positioning right and Y-axis pointing forwards with respect to the robot front side as depicted in Figure 5.9.

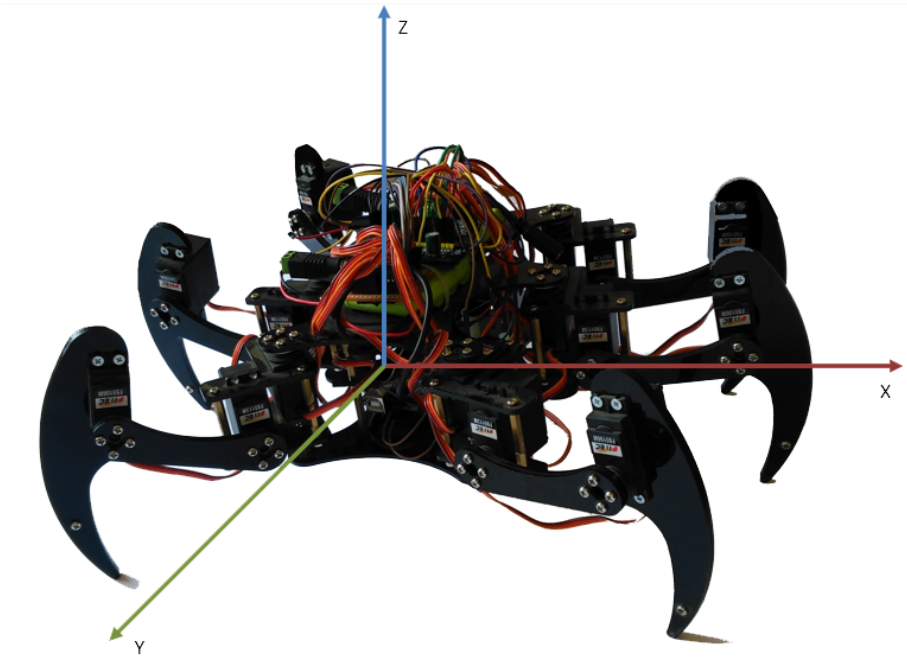

Figure 5.9.: Location of body frame relative to robot hardware.

5.3.2. Leg frames and notations

The design of hexapod constitutes the kinematic configuration of a hexapod robot, with each leg acting as an independent serial manipulator with three degrees of freedom. Figure Figure 1.1 shows the actual prototype of our robot.

The final leg design and its links and joints notations are given in Figure 5.10. The robot leg is made of links and joints as noted on Figure 5.11, different links of robot leg are called Femur, Tibia and Tarsus. As depicted in figure, the robot leg frame starts with link 0, which is the point where the leg is attached to the body, link 1, is Femur, link 2 is the Tibia and link 3 is Tarsus. The joints are located at the inner end of their respective link. Frames are attached to outer end of their respective links, this means that joint 2 rotates about the Z-axis of frame 1.

5.3.3. Robot Leg Parameters

Following the well-known Denavit-Hartenberg (DH) notation, coordinate frames for the robot leg are assigned. The assigned frames are shown in Figure 5.11. In figure, the body b and the zeroth 0 reference frames are attached to the stationary robot body. Therefore, they can be both considered as inertial frames. The axes of the body frame are arranged to be in accord with the actual robot-body orientation. The DH link parameters based on Figure 5.11 are given in Table II.

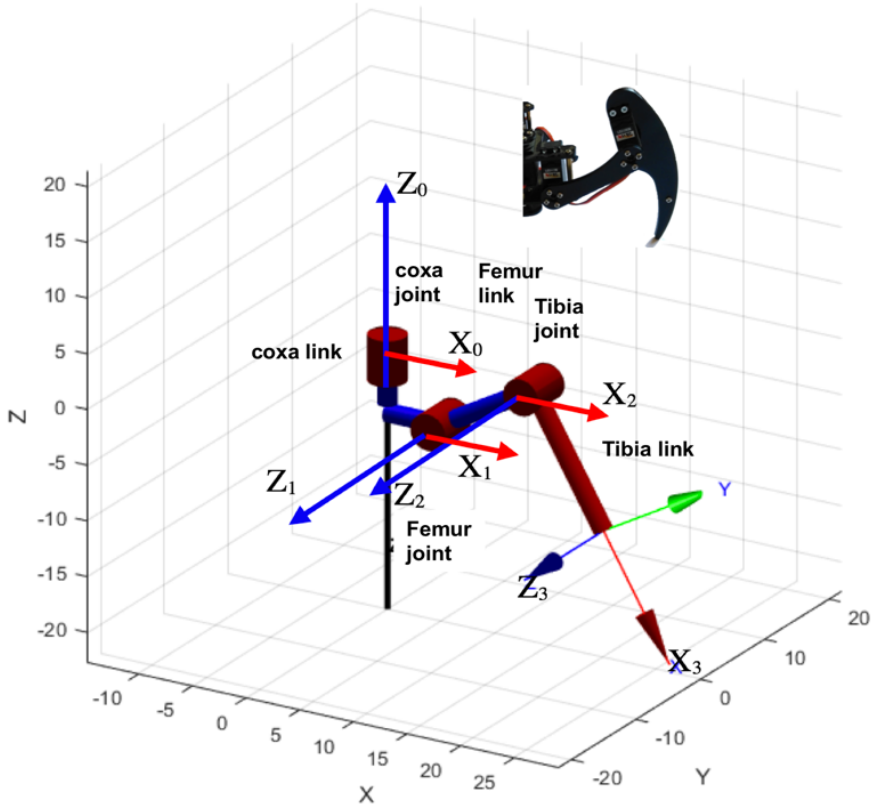

Figure 5.10.: Final leg design (top right) and its notations, reference frames, joints and links.

The resulting homogeneous transformation matrices between the body and the zeroth frame and between the sequential link frames are given in (1). In the formulas, the variables represented by a stand for the length of the i^{th} link (namely, the length of the portion of the link between the origins of $(i-1)thandi^{th}$ reference frames). The variables represented by θ_{ij} mean the sum of the i^{th} and j^{th} joint angles ($\theta_{ij} = \theta_i + \theta_j$). C and S are for cos(.) and sin(.) functions, respectively. The exact values of these variables corresponding to ZagHexa robot are: $\psi = 45°, a_1 = 5cm, a_2 = 9cm, a_3 = 18cm$

Joint	θ_i	α_i	a_i	d_i
1	θ_1	$\pi/2$	a_1	0
2	θ_2	0	a_2	0
3	θ_3	0	a_3	0

Homogeneous matrices are used in derivation of positional relations between the successive frames. In (3) the leg tip point position with respect to the body frame is given. The rotation matrices between the frames are given in (2). These rotation matrices are used in vector equations, especially while deriving the dynamic equations.

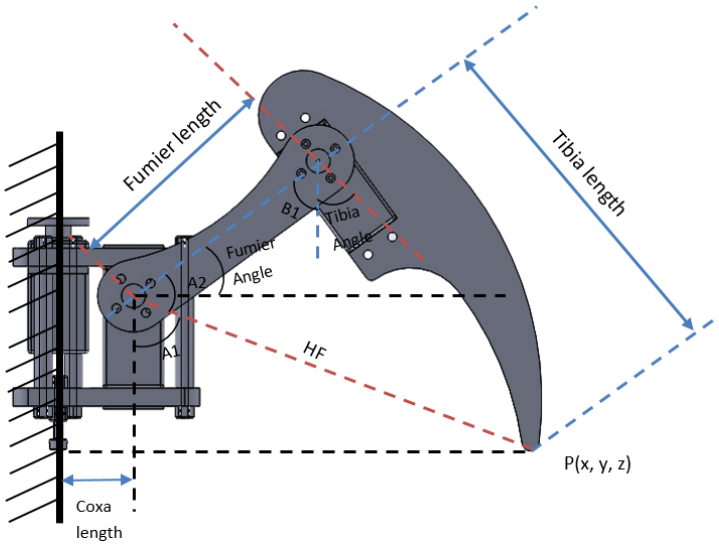

Figure 5.11.: Final leg design (top right) and its notations, reference frames, joints and links.

$$H^{(b,0)} = \begin{bmatrix} 0 & \cos(\psi) & \sin(\psi) & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H^{(K-1,K)} = \begin{bmatrix} \cos\theta_k & -\cos\alpha_k \sin\theta_k & \sin\alpha_k \sin\theta_k & a_k \cos\theta_k \\ \sin\theta_k & \cos\alpha_k \cos\theta_k & -\sin\alpha_k \cos\theta_k & a_k \sin\theta_k \\ 0 & \sin\alpha_k & \cos\alpha_k & d_k \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

$$H^{(0,3)} = H^{(0,1)} H^{(1,2)} H^{(2,3)} H^{(b,3)} = H^{(b,0)} H^{(0,3)}$$

$$C^{(b,0)} = \begin{bmatrix} 0 & \cos(\psi) & \sin(\psi) \\ -1 & 0 & 0 \\ 0 & -\sin(\psi) & \cos(\psi) \end{bmatrix}$$

$$C^{(K-1,K)} = \begin{bmatrix} \cos\theta_k & -\cos\alpha_k \sin\theta_k & \sin\alpha_k \sin\theta_k \\ \sin\theta_k & \cos\alpha_k \cos\theta_k & -\sin\alpha_k \cos\theta_k \\ 0 & \sin\alpha_k & \cos\alpha_k \end{bmatrix} \tag{5.2}$$

$$P_e^{(K-1,K)}(\theta) = \begin{bmatrix} C\psi(a_1 S\theta_1 + a_2 S\theta_1 C\theta_2 + a_3 S\theta_1 C\theta 23) + S\psi(a_2 S\theta_2 + a_3 \theta_{23}) \\ -(a_1 C\theta_1 + a_2 C\theta_1 C\theta_2 + a_3 C\theta_1 C\theta_{23}) \\ -S\psi(a_1 S\theta_1 + a_2 S\theta_1 C\theta_2 + a_3 S\theta_1 C\theta 23) + C\psi(a_2 S\theta_2 + a_3 \theta_{23}) \end{bmatrix} \tag{5.3}$$

To derive the dynamic equations, first the inertia matrices of the links should be determined. Since the k^{th} reference frame is stationary with respect to the k^{th} link, the inertia tensor of the k^{th} link around its center of mass appears to be a constant matrix with respect to the k^{th} reference frame, as in (4). The values used in these formulations belong to the Hexapod robot. The resulting matrices for each link are in the form of (5).

$$\{J_K\}^{(K)} = J_K^{(K)} = J_K \tag{5.4}$$

$$J_K = \begin{bmatrix} J_{K1} & 0 & 0 \\ 0 & J_{K2} & 0 \\ 0 & 0 & J_{K3} \end{bmatrix} \tag{5.5}$$

5.3.4. Inverse kinematics

The forward kinematics (FK) is a simple equation used to calculate the position of the end effectors for the leg in the robot base frame, by injecting values of each joint angle. But the reverse operation, namely inverse kinematics (IK), is more complex. IK is employed to find all the joint angles given the position of the end effectors. In general, solving the IK equations can be a bit of a challenge. Some positions cannot be reached at all, as the physical system is unable to get there, and some end effectors positions can have more than one solution, and not all of them are desirable.

We solve the IK problem for each leg separately, as this makes it possible to solve it geometrically, by setting up some constrains. The first constraint for solving the IK equations due the fact that all robot joints allow rotation about one axis only. The second constraint is that the Femur, Tibia joints always rotate on parallel axes. The third set of constraints arises from the physical limitations for each joint, giving us some angular interval for each joint in which the servos can actually rotate the link. In Figure 5.11, the angles of movement are shown. First, the coxa angle can be found directly by knowing the end effectors position then simply using $atan2(y,x)$ to calculate it. Equations (6) through (14) are used to find the individual joint angles.

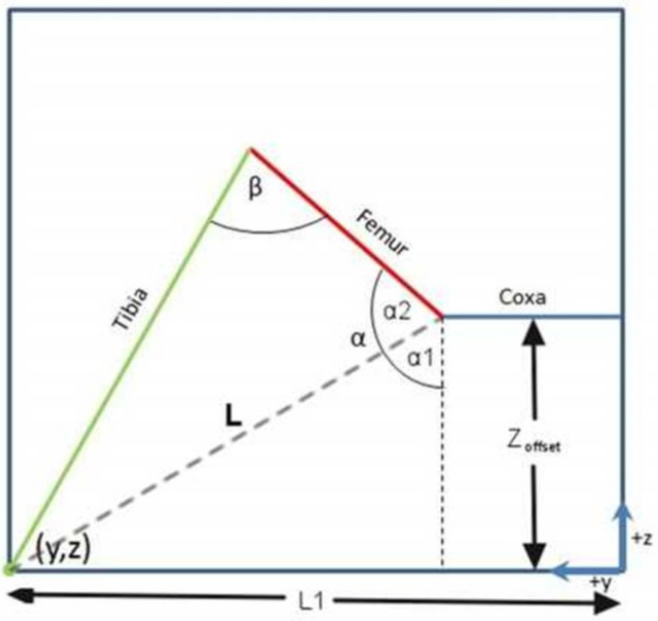

Figure 5.12.: Illustration of the 2D triangle with vertices in the coxa, the femur, and tibia link from origin.

$$\frac{x}{y} = \tan(y) \rightarrow \gamma = \tan^{-1}\frac{x}{y} \tag{5.6}$$

$$L = \sqrt{Z_{offset}^2 + (L_1 + \cos A)^2} \tag{5.7}$$

$$a_L = \cos^{-1}(\frac{Z_{offset}}{L}) \tag{5.8}$$

$$Tibia^2 = Femar^2 + L^2 - 2(Femar)(L)\cos\alpha_2 \tag{5.9}$$

$$\alpha_2 = \cos^{-1}(\frac{Tibia^2 - Femar^2 - L^2}{-2(Femar)(L)}) \tag{5.10}$$

$$\alpha = \alpha_1 + \alpha_2 \tag{5.11}$$

$$\alpha = \cos^{-1}(\frac{Z_{offset}}{L}) + \cos^{-1}(\frac{Tibia^2 - Femar^2 - L^2}{-2(Femar)(L)}) \tag{5.12}$$

$$\beta = \cos^{-1}(\frac{L^2 - Femar^2 - Tibia^2}{-2(Femar)(Tibia)} \tag{5.13}$$

"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong."

— Richard P. Feynman, (American theoretical physicist, 1918–1988)

Chapter 6

Experiments and Simulation

Programming directly on a real robot gives us good feedback and it is more impressive than simulations, but not everybody has possible access to real robots. For this reason, we have programs that simulate the physical world.
The first phase of robot manufacturing is its design and modeling. We can design and model the robot using CAD tools such as Solid Works, Blender, and so on. One of the main purposes of modeling robot is simulation. The robotic simulation tool can check the critical flaws in the robot design and can confirm the working of the robot before it goes to the manufacturing phase.

The virtual robot model must have all the characteristics of real hardware, the shape of robot may or may not look like the actual robot but it must be an abstract, which has all the physical characteristics of the actual robot.

If we are planning to create the 3D model of the robot and simulate using ROS, you need to learn about some ROS packages which helps in robot designing. ROS has a standard meta package for designing, and creating robot models called robot model, which consists of a set of packages called urdf, robot state publisher and so on. These packages help us create the 3D robot model description with the exact characteristics of the real hardware.

In this chapter, we will cover the following topics:

1. ROS packages for robot modeling
2. Understanding robot modeling using URDF
3. Creating our URDF model
4. Watching the 3d model in RVIZ
5. Making our robot movable

6.1. ROS packages for robot modeling

The way ROS uses the 3D model of a robot or its parts, to simulate them. ROS provides some good packages that can be used to build 3D robot models.
In this section, we will discuss some of the important ROS packages that are commonly used to build robot models:

robot model: ROS has a meta package called robot model, which contains important packages that help build the 3D robot models. We can see all the important packages inside this meta- package:

URDF: One of the important packages inside the robot model meta package is urdf. The URDF package contains a C++ parser for the Unified Robot Description Format (URDF), which is an XML file to represent a robot model.

We can define a robot model, sensors, and a working environment using URDF and can parse it using URDF parsers.
We can only describe a robot in URDF that has a tree-like structure in its links, that is, the robot will have rigid links and will be connected using joints. Flexible links can't be represented using URDF.
The URDF is composed using special XML tags and we can parse these XML tags using parser programs for further processing. We can work on URDF modeling in the upcoming sections.

joint state publisher: This tool is very useful while designing robot models using URDF.
This package contains a node called joint state publisher, which reads the robot model description, finds all joints, and publishes joint values to all non fixed joints using GUI sliders.
The user can interact with each robot joint using this tool and can visualize using RViz. While designing URDF, the user can verify the rotation and translation of each joint using this tool.

kdl parser: Kinematic and Dynamics Library (KDL) is an ROS package that contains parser tools to build a KDL tree from the URDF representation. The kinematic tree can be used to publish the joint states and also to forward and inverse kinematics of the robot.

robot state publisher: This package reads the current robot joint states and publishes the 3D poses of each robot link using the kinematics tree build from the URDF. The 3D pose of the robot is published as ROS tf (transform). ROS tf publishes the relationship between coordinates frames of a robot.

xacro: Xacro stands for (XML Macros) and we can define how xacro is equal to URDF plus add-ons. It contains some add-ons to make URDF shorter, readable, and can be used for building complex robot descriptions. We can convert xacro to URDF at any time using some ROS tools. We will see more about xacro and its usage in the upcoming sections.

6.2. Understanding robot modeling using URDF

We have discussed the urdf package. In this section, we will look further at the URDF XML tags, which help to model the robot. We have to create a file and write the relationship between each link and joint in the robot and save the file with the .urdf extension.
The URDF can represent the kinematic and dynamic description of the robot, visual representation of the robot, and the collision model of the robot.

The following tags are the commonly used URDF tags to compose a URDF robot model:
link: The link tag represents a single link of a robot. Using this tag, we can model a robot link and its properties. The modeling includes size, shape, color, and can even import a 3D mesh to represent the robot link. We can also provide dynamic properties of the link such as inertial matrix and collision properties.
The syntax is as follows:

```
<link name="<name of the link>">
    <inertial>...........</inertial>
    <visual> ............</visual>
    <collision>..........</collision>
</link>
```

The following is a representation of a single link. The Visual section represents the real link of the robot, and the area surrounding the real link is the Collision section. The Collision section encapsulates the real link to detect collision before hitting the real link.
joint: The joint tag represents a robot joint. We can specify the kinematics and dynamics of the joint and also set the limits of the joint movement and its velocity. The joint tag supports the different types of joints such as revolute, continuous, prismatic,fixed, floating, and planar.

The syntax is as follows:

```
<joint name="<name of the joint>">
  <parent link="link1"/>
  <child link="link2"/>
  <calibration .... />
  <dynamics damping ..../>
  <limit effort .... />
</joint>
```

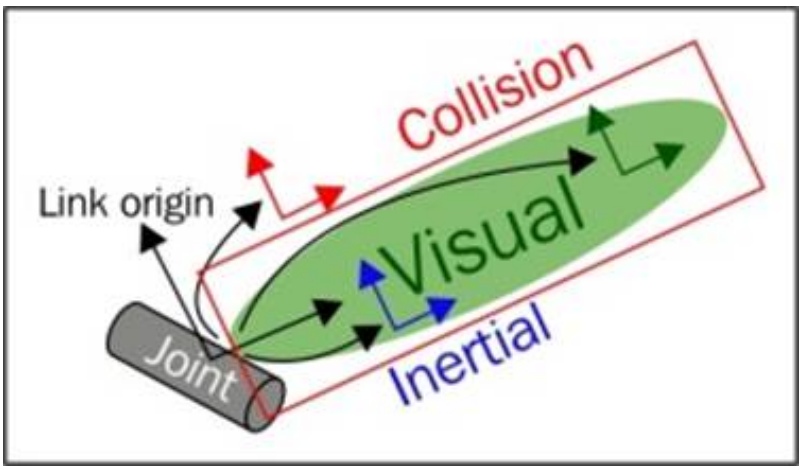

Figure 6.1.: Visualization of a URDF link

A URDF joint is formed between two links; the first is called the Parent link and the second is the Child link. The following is an illustration of a joint and its link:
robot: This tag encapsulates the entire robot model that can be represented using

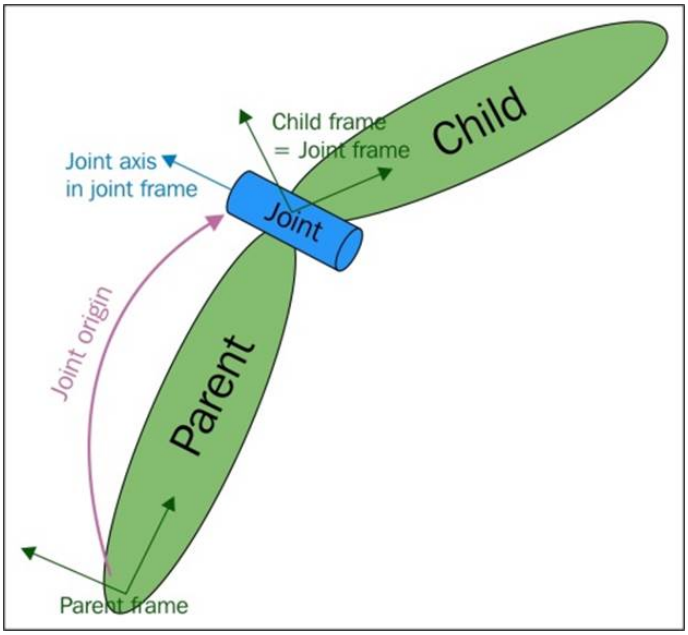

Figure 6.2.: Visualization of a URDF joint

URDF. Inside the robot tag, we can define the name of the robot, the links, and the joints of the robot. The syntax is as follows:

```
<robot name="<name of the robot>"
  <link> ..... </link>
  <link> ..... </link>
  <joint> ..... </joint>
```

```
  <joint> ..... </joint>
</robot>
```

A robot model consists of connected links and joints. Here is a visualization of the robot model:

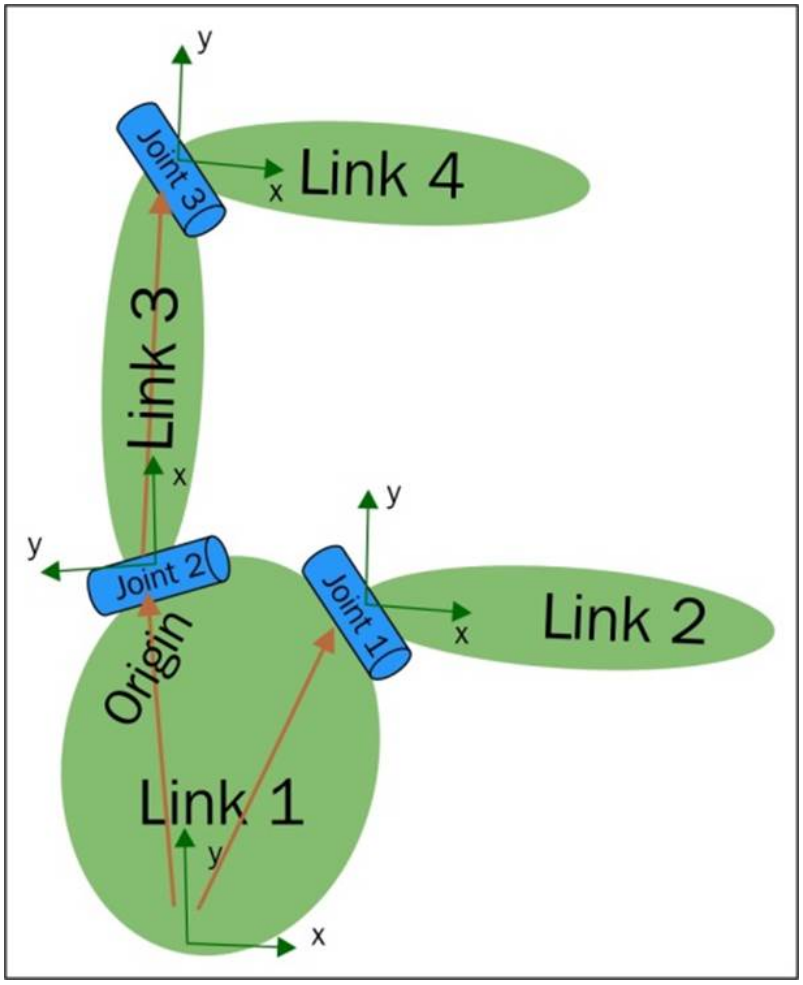

Figure 6.3.: Visualization of a robot model having joints and links

6.3. Creating our URDF model

To start first we create the urdf file, let's call it *zaghexasim.urdf* and put in the following code; this URDF code is based on XML.As you will see in the code, there are two principal fields that describe the geometry of a robot: links and joints.
the first link has the name base link; this name must be unique to the file

```
<?xml version="1.0" ?>
<robot name="zaghexa" xmlns:xacro="http://ros.org/wiki/xacro">
 <!-- Build the body frame -->
<link name="base_link"/>
<joint name="base_joint" type="fixed">
<parent link="base_link"/>
<child link="box"/>
<origin rpy="0 0 0" xyz="0 0 0"/>
</joint>
<link name="box">
<visual>
<origin rpy="0 0 0" xyz="0 0 0"/>
<geometry>
<mesh filename="package://zaghexa_sim/meshes/box.STL"/>
</geometry>
<material name="grey">
<color rgba="0.5 0.5 0.5 1"/>
</material>
</visual>
</link>
```

In the joint field we define the name which must be unique as well also we define the type of joint(fixed,revolute,continous,floating or planar) the parent, and the child. in our case tibia,femur and leg centre joint are the children of base link which is fixed but all of other joints are revolute.
this is a sample of one leg and how does it build

```
<!-- Joint properties -->
<!-- Leg macros -->
<!-- Build robot model -->
<joint name="leg_center_joint_r1" type="fixed">
<origin rpy="0 0 0" xyz="0.087598 -0.050575 0"/>
<parent link="box"/>
<child link="leg_center_r1"/>
</joint>
<link name="leg_center_r1"/>
<joint name="coxa_joint_r1" type="revolute">
<origin rpy="0 0 -1.0471975512" xyz="0 0 0"/>
<parent link="leg_center_r1"/>
<child link="coxa_r1"/>
<axis xyz="0 0 -1"/>
<limit effort="10000" lower="-1.5" upper="1.5" velocity="100"/>
</joint>
<link name="coxa_r1">
<visual>
<origin rpy="0 0 0" xyz="0 0 0"/>
<geometry>
<mesh filename="package://zaghexa_sim/meshes/coxa_r.STL"/>
</geometry>
<material name="">
<color rgba="0.7 0.7 0 1"/>
```

```
</material>
</visual>
</link>
<joint name="femur_joint_r1" type="revolute">
<origin rpy="-1.57079632679 0 0" xyz="0.0294 0 0"/>
<parent link="coxa_r1"/>
<child link="zaghexa"/>
<axis xyz="0 0 -1"/>
<limit effort="10000" lower="-1.5" upper="1.5" velocity="100"/>
</joint>
<link name="zaghexa">
<visual>
<origin rpy="0 0 0" xyz="0 0 0"/>
<geometry>
<mesh filename="package://zaghexa_sim/meshes/femur_r.STL"/>
</geometry>
<material name="">
<color rgba="0 0.7 0.7 1"/>
</material>
</visual>
</link>
<joint name="tibia_joint_r1" type="revolute">
<origin rpy="3.14159265359 0 1.57079632679" xyz="0.08 0 0"/>
<parent link="zaghexa"/>
<child link="tibia_r1"/>
<axis xyz="0 0 1"/>
<limit effort="10000" lower="-1.5" upper="1.5" velocity="100"/>
</joint>
<link name="tibia_r1">
```

You can check the syntax of the urdf whether we have errors, we can use: check urdf command tool:

```
$ rosrun urdf_parser check_urdf zaghexa_sim.urdf
```

If you want to see it graphically, you can use the urdf to graphiz command tool

```
$ rosrun urdf_parser urdf_to_graphiz "`rospack find zaghexa_sim`/
    urdf/zaghexa_sim.urdf"
```

The following is what you will receive as output:

6.4. Watching the 3D model in RVIZ

Now that we have the model of our robot, we can use it on rviz to watch it in 3D and see the movements of the joints.
We will create the display.launch file in zaghexa-sim/launch folder, and put the following code in it:

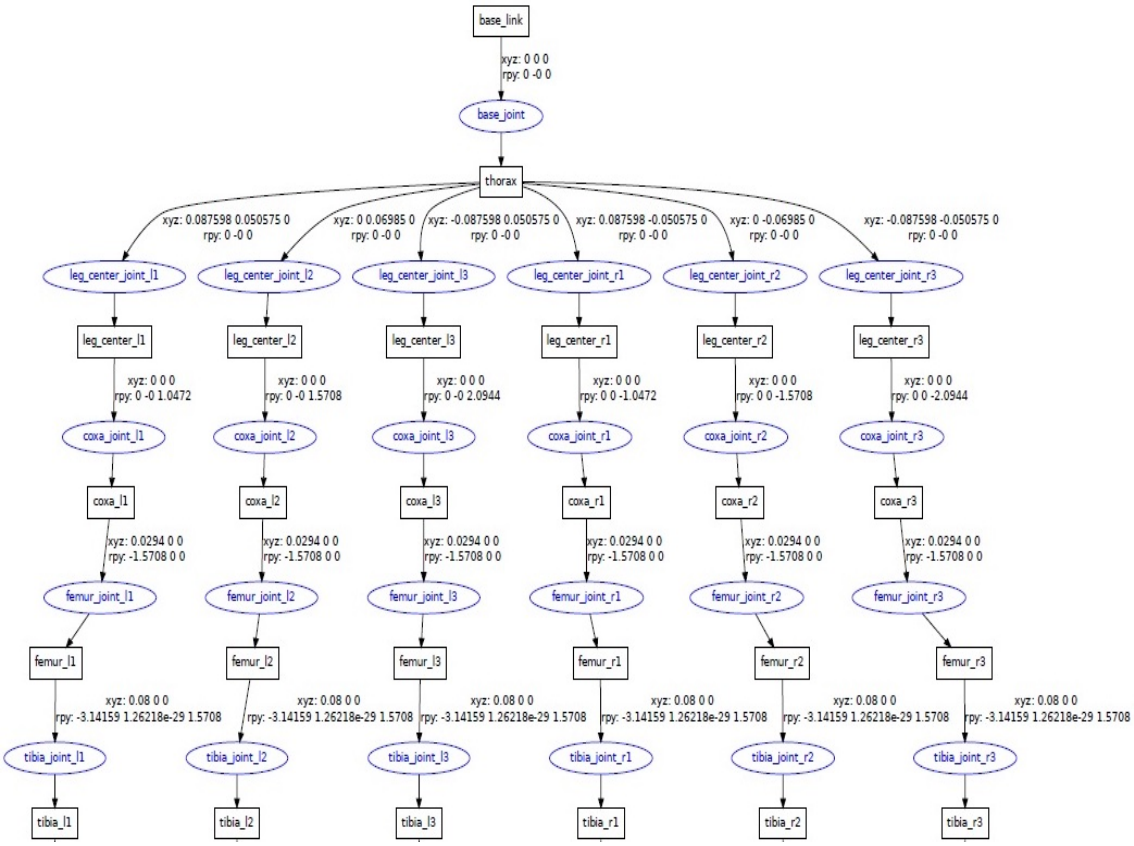

Figure 6.4.: output of urdf to graphics

```
>>>>>>> origin/master
        <launch>
        <arg
        name="model" />
        <arg
        name="gui"
        default="True" />
        <param
        name="robot_description"
        command="$(find xacro)/xacro.py '$(find zaghexa_sim)/models/zaghexa_model.
            xacro'" />
        <param
        name="use_gui"
        value="$(arg gui)" />
        <param
        name="rate"
        value="25" />
        <rosparam param="source_list">
        [leg_joints_states]
        </rosparam>
        <node
```

```
        name="joint_state_publisher"
        pkg="joint_state_publisher"
        type="joint_state_publisher" />
        <node
        name="robot_state_publisher"
        pkg="robot_state_publisher"
        type="state_publisher" />
        <node
        name="rviz"
        pkg="rviz"
        type="rviz"
        args="-d $(find zaghexa_sim)/urdf.rviz" />
        </launch>
<<<<<<< HEAD
```

We will launch it with the following command:

```
$ roslaunch zaghexa_sim display_model.launch model:="‘
    rospack find
zaghexa_sim‘/urdf/zaghexa_sim.urdf"
```

if every thing is fine and you have no errors, it will load RVIZ and you will see:

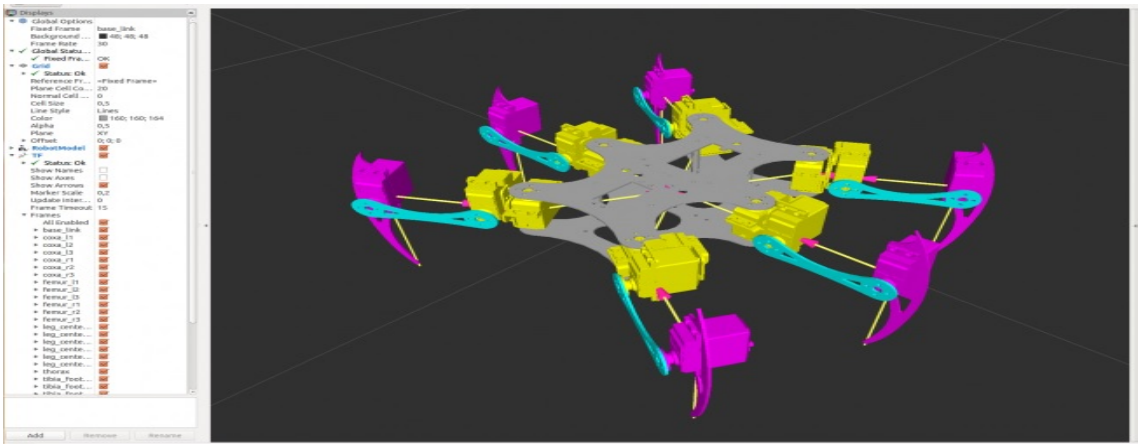

Figure 6.5.: output of urdf to graphics

6.4.1. Making our robot movable

A good way of testing whether or not the axis and limits of the joints are fine by running rviz with joint state publisher GUI

```
$ roslaunch zaghexa_sim display.launch model:="‘rospack
    find
zaghexa_sim‘/urdf/zaghexa_sim.urdf" gui:=true
```

you will see a GUI with some sliders each of them controls one joint of the 18 joints so we have 18 sliders:

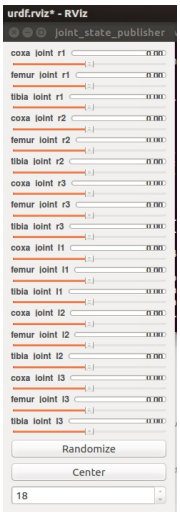

Figure 6.6.: Joint state publisher GUI

In the next figures you will see the effect of changing sliders values to the joints angles and positions

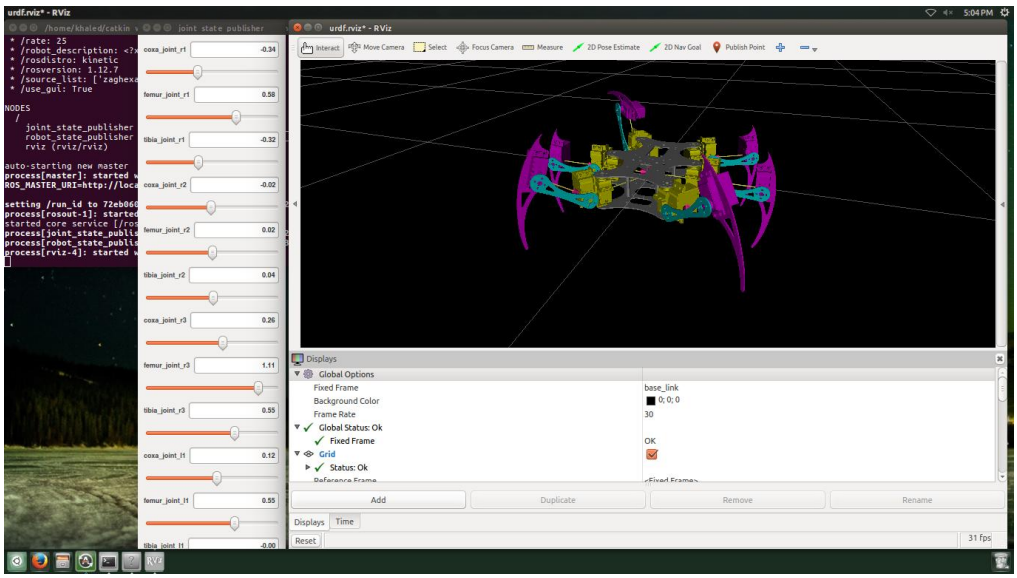

Figure 6.7.: Control sliders and their effect on the robot

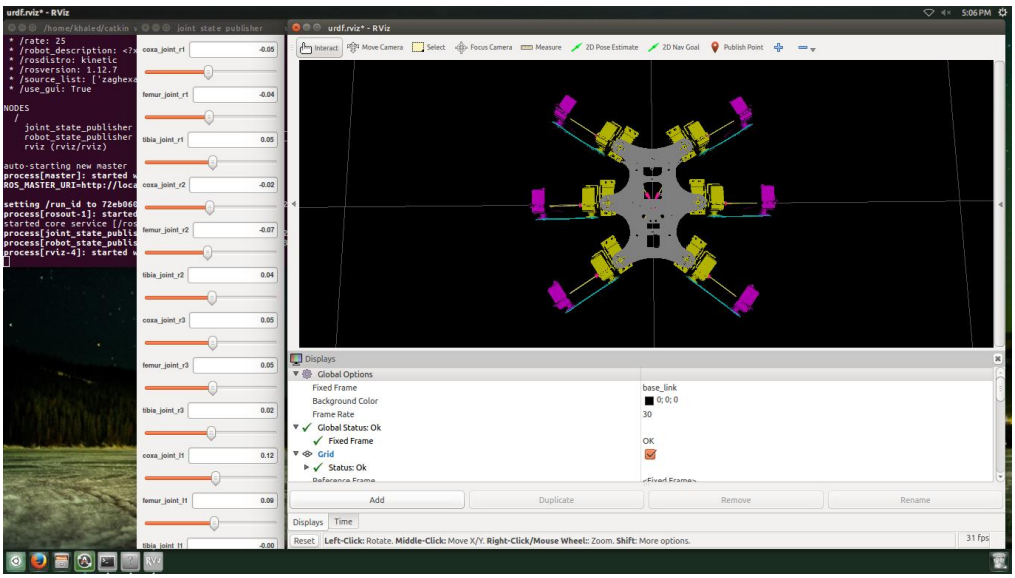

Figure 6.8.: top view of the robot

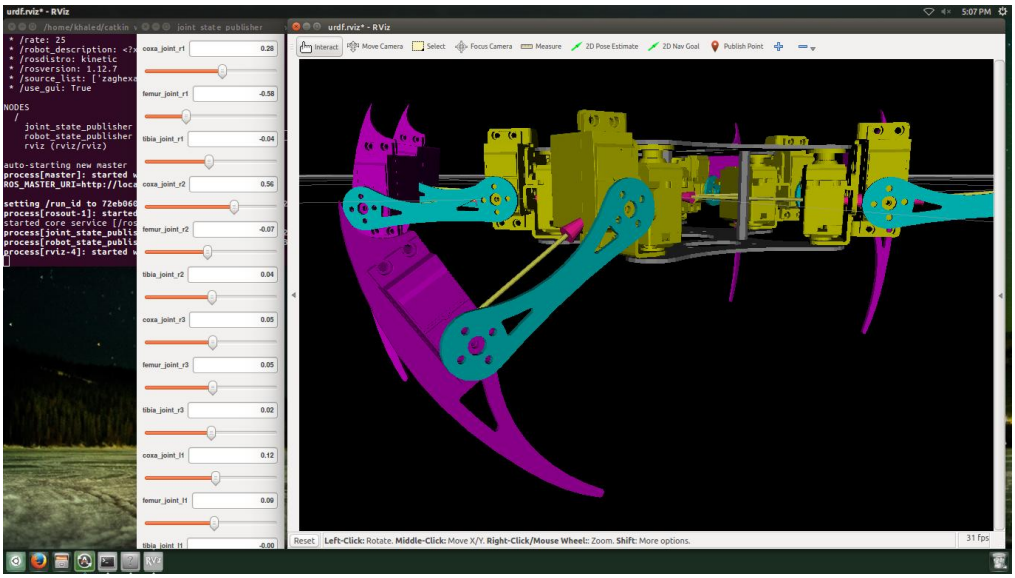

Figure 6.9.: Different views of the robot

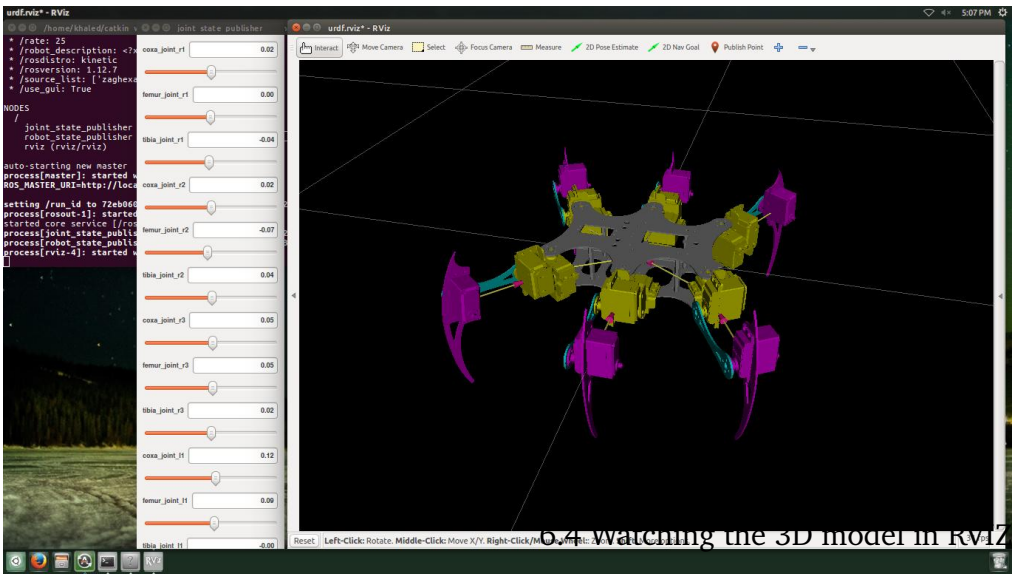

Figure 6.10.: Different views of the robot

6.5. Matlab Simulation

In the second simulation, we modelled the robot in MATLAB and employing the Robotics Toolbox. The main purpose of this simulation is to calculate and simulate the kinematics of robot. To create the six-legged walking robot, we started by creating a three-axis robot arm that we used as a leg. Then we implemented a trajectory for the leg that is suitable for walking. Finally, we instantiated six instances of the leg to create the walking robot. The equations given in Sec.4 are programmed first for one leg and tested on successful working, the whole body kinematics were also programmed and tested. The results were very useful in modifying the walking gaits of the robot which then implemented in the real robot. Figure 10 shows one such simulations in which the same experiment performed on the real robot to test the whole body kinematics for raising and lowering the body height.

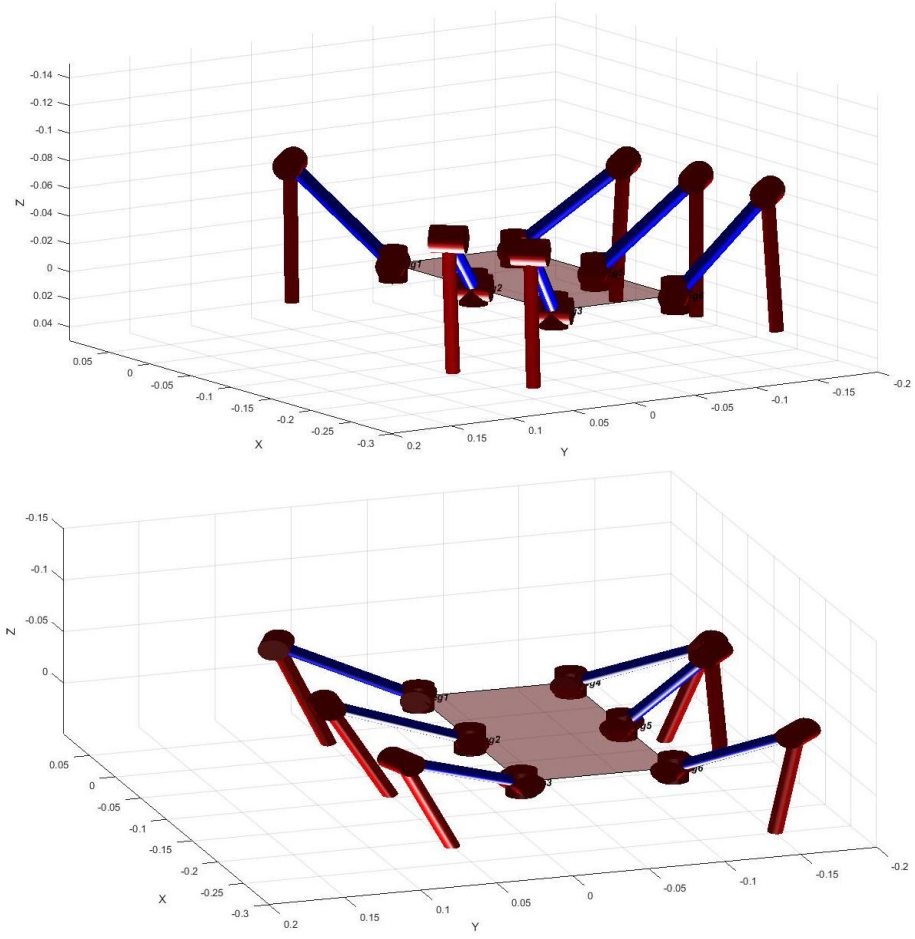

Figure 6.11.: Hexapod robot Simulation.

We will launch it with the following command:

```
$ roslaunch zaghexa_sim display_model.launch model:="`rospack find
    zaghexa_sim`/urdf/zaghexa_sim.urdf"
```

if every thing is fine and you have no errors, it will load RVIZ and you will see:

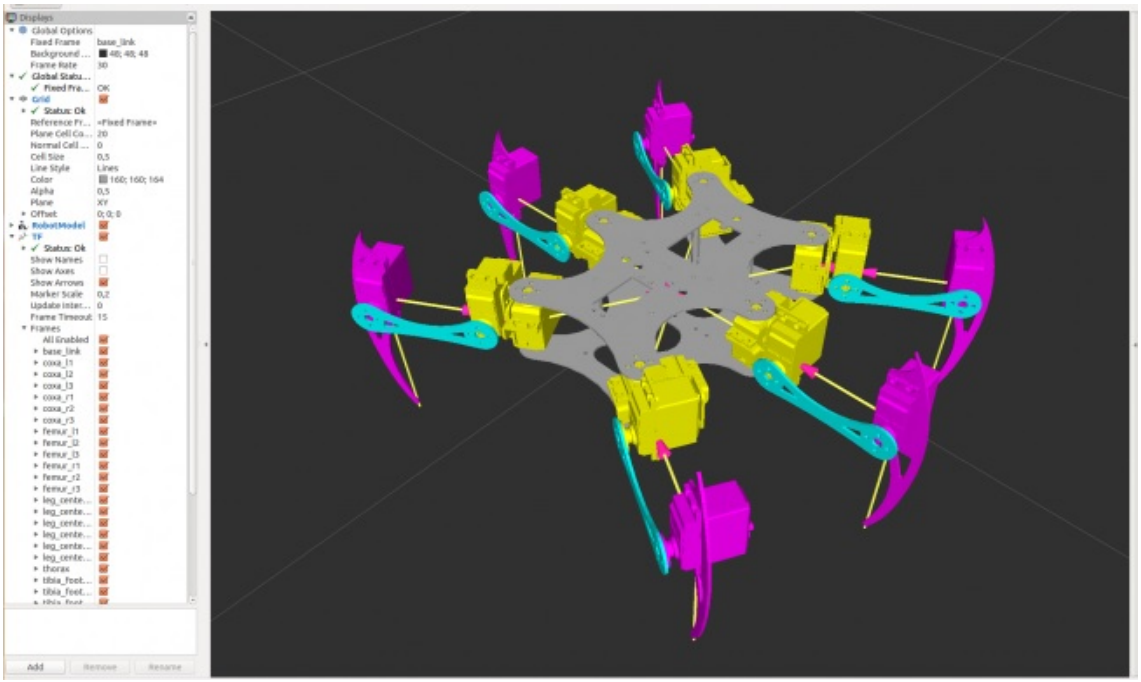

Figure 6.12.: output of urdf to graphics

6.6. Making our robot movable

A good way of testing whether or not the axis and limits of the joints are fine by running rviz with joint state publisher GUI

```
$ roslaunch zaghexa_sim display.launch model:="‘rospack find
    zaghexa_sim‘/urdf/zaghexa_sim.urdf" gui:=true
```

you will see a GUI with some sliders each of them controls one joint of the 18 joints so we have 18 sliders:
In the next figures you will see the effect of changing sliders values to the joints angles and positions

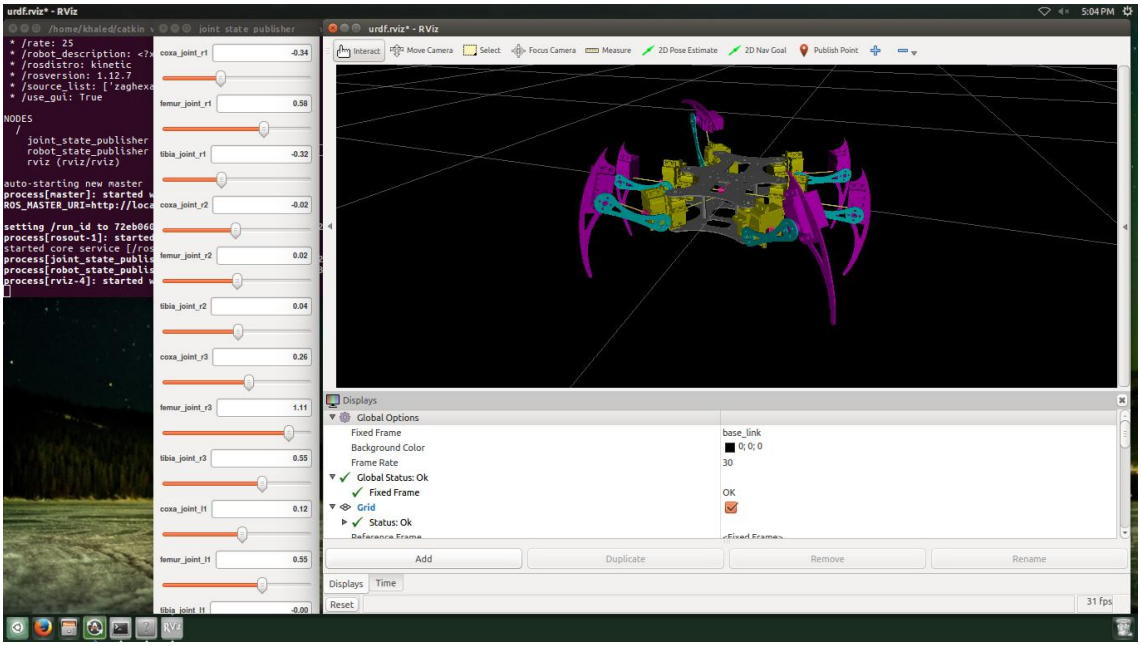

Figure 6.13.: Joint state publisher GUI with its control sliders and their effect on the robot

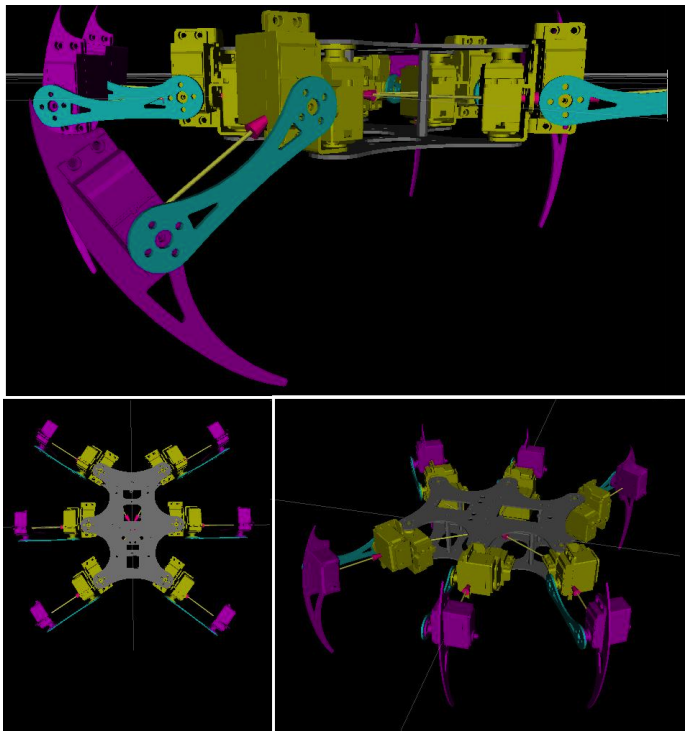

Figure 6.14.: Different views of the robot

The true function of philosophy is to educate us in the principles of reasoning and not to put an end to further reasoning by the introduction of fixed conclusions.

— George Henry Lewes, (English philosopher and critic of literature, 1817–1878)

Chapter 7
Conclusions and Future Outlook

This paper presents a system description and the main aspects related to the design, construction, and implementation of six-leg robot named ZagHexa. The robot is a legged robot for search and rescue missions. It benefits form the reliability of its legged locomotion with the flexibility and versatility required to operate in different types of surface. The robot was constructed and tested to walk using tripod, wave and ripple gaits, can rotate and it is equipped with different sensors.

The robot was tested on different surfaces and in rugged terrain. The repeatability of the robot movement as well as the sensor system was also tested. These features are mainly achieved due to its original movement that make it deal with different surfaces. Additionally, its shape and weight give it more stability, and its ability to continue with its moving and sensing capabilities after collisions or even small falls.
However, more tests and experiments to improve and validate the design and sensor performance are to be carried out to optimize the system performance. Finally, we are working on tackling some issues should to have fully autonomous operation and integration into a heterogeneous system. To make the integration of ZagHexa into different missions easier, an effort is being carried to provide it with a standard connectivity over the ROS framework.

Appendix A

Introduction to ROS

The Robot Operating System (ROS) is a software framework for developing robotic systems in a metaoperating system environment. The primary goal of ROS is to support code reuse in robotics research and development.

ROS is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. ROS provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, messagepassing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

ROS is comprised of a number of independent nodes, each of which communicates with the other nodes using a publish /subscribe messaging model "we will explain this soon". ROS was originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and development continued at Willow Garage. It is managed by the Open Source Robotics Foundation.

Ready to get started with ROS ? You've come to the right place. Here you will find our collection of step-by-step tutorials.

First, you must know that ROS is a framework which only runs on Unix-based platforms" We used Ubuntu". We used ROS Kinetic Kame distribution, which is available for Ubuntu Xenial(16.04 LTS), among other platform options.

A.1. Prerequisites to start with ROS

Before getting started with ROS and trying code, the following prerequisites should be met:

First, We have to use Ubuntu as the operating system for installing ROS. Its prefer to stick on to the L.T.S version of Ubuntu, that is, `www.releases.ubuntu.com/16.04.2/` Ubuntu 16.04.2 LTS (Xenial Xerus).
Hint, select an image which is suitable for you, then burn it into removable storage with any software like rufus, and the installation will done using the GUI guide. If you don't use ubuntu operating system or any distribution of unix before, Don't worry, we will explain the concepts at the ubuntu operating system section. Now, Install the full desktop installation of ROS. The following link gives you the installation instruction of the L.T.S ROS distribution: `http://wiki.ros.org/kinetic/Installation/Ubuntu`.
Hint, Just follow the installation instructions which is copying the commands from the website and paste it in the command line window on your PC.

A.2. General Concepts

The basic building blocks of Robot Operating System "ROS software framework are Packages. ROS works as a group of programs everyone is called Node. Every Package contains a collection of Nodes. ROS starts with the ROS Master node. The Master node allows all other ROS pieces of software (Nodes) to find and talk to each other. Every Node communicate with each other through Messages.

- **Packages**:
  The ROS packages are the most basic unit of the ROS software. It contains the ROS runtime process (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build item and release item in the ROS software.
- **Package manifest**: The package manifest file is inside a package that contains information about the package, author, license, dependencies, compilation flags, and so on. The package.xml file inside the ROS package is the manifest file of that package.

A.2.1. Structure of a typical ROS package

package.xml: This is the package manifest file of this package.
CMakeLists.txt: This is the CMake build file of this package.
msg: This folder contains custom message definitions.
srv: This folder contains the service definitions.

include/package_name: This folder consists of headers and libraries that we need to use inside the package.
scripts: This folder keeps executable Python scripts.
src: This folder stores the C++ source codes.
We need to know some commands to create, modify, and work with the ROS packages. Here are some of the commands used to work with ROS packages:

- **catkin_create_pkg**: This command is used to create a new package.
- **rospack**: This command is used to get information about the package in the file system.
- **catkin_make**: This command is used to build the packages in the workspace.
- **rosdep**: This command will install the system dependencies required for this package.

A.2.2. Messages (.msg)

The ROS messages are a type of information that is sent from one ROS process to the other, we can define a custom message inside the msg folder inside a package (my_package/msg/ MyMessageType.msg), the extension of the message file is .msg, ROS nodes can publish data having a particular type and communicate with each other using messages. The types of data are described using a simplified message description language, also called ROS messages, these datatype descriptions can be used to generate source code for the appropriate message type in different target languages.

Messages are simply a data structure containing the typed field, which can hold a set of data and that can be sent to another node. There are standard primitive types (integer, floating point, Boolean, and so on) and these are supported by ROS messages. We can also build our own message types using these standard types.

Here are some parameters used along with rosmsg:

- rosmsg show [message]: This shows the message description.
- rosmsg list: This lists all messages.
- rosmsg package [package_name]: This lists messages in a package.
- rosmsg packages [package_1] [package_2]: This lists packages that contain messages.

A.2.3. Services (.srv)

The ROS service is a kind of request/reply interaction between processes (Nodes), one node can send a request and wait until it gets a response from the other. The

request/response communication is also using the ROS message description. The reply and request data types can be defined inside the srv folder inside the package (my_package/srv/MyServiceType.srv).

In some robot applications, a publish/subscribe model will not be enough if it needs a request/response interaction. The publish/subscribe model is a kind of one-way transport system and when we work with a distributed system, we might need a request/response kind of interaction.

ROS Services are used in these case. We can define a service definition that contains two parts; one is for requests and the other is for responses. Using ROS Services, we can write a server node and client node. The server node provides the service under a name, and when the client node sends a request message to this server, it will respond and send the result to the client. The client might need to wait until the server responds. The ROS service interaction is like a remote procedure call.

The following explain how to use the rosservice tool to get information about the running services:

- rosservice call /service args: This tool will call the service using the given arguments.
- rosservice find service_type: This command will find services in the given service type.
- rosservice info /services: This will print information about the given service.
- rosservice list: This command will list the active services running on the system.
- rosservice type /service: This command will print the service type of a given service.
- rosservice uri /service: This tool will print the service ROSRPC URI.

A.2.4. Topics

Each message in ROS is transported using named buses called topics. When a node sends a message through a topic, then we can say the node is publishing a topic. When a node receives a message through a topic, then we can say that the node is subscribing to a topic. The publishing node and subscribing node are not aware of each other's existence. We can even subscribe a topic that might not have any publisher. In short, the production of information and consumption of it are decoupled. Each topic has a unique name, and any node can access this topic and send data through it as long as they have the right message type.

The ROS topic tool can be used to get information about ROS topics. Here is the syntax of this command:

- rostopic bw /topic: This command will display the bandwidth used by the given topic.

- rostopic echo /topic: This command will print the content of the given topic.
- rostopic find /message_type: This command will find topics using the given message type.
- rostopic hz /topic: This command will display the publishing rate of the given topic.
- rostopic info /topic: This command will print information about an active topic.
- rostopic list: This command will list all active topics in the ROS system.
- rostopic pub /topic message_type args: This command can be used to publish a value to a topic with a message type.
- rostopic type /topic: This will display the message type of the given topic.

A.2.5. Nodes

Nodes are the process that perform computation. Each ROS node is written using ROS client libraries such as roscpp and rospy. In a robot, there will be many nodes to perform different kinds of tasks. Using the ROS communication methods, it can communicate with each other and exchange data. One of the aims of ROS nodes is to build simple processes rather than a large process with all functionality. Being a simple structure, ROS nodes are easy to debug too.

One node can communicate with other nodes using ROS Topics, Services, and Parameters. A robot might contain many nodes, for example, one node processes camera images, one node handles serial data from the robot, and so on.

Using nodes can make the system fault tolerant. Even if a node crashes, an entire robot system can still work. Nodes also reduce the complexity and increase debug-ability compared to monolithic codes because each node is handling only a single function.

All running nodes should have a name assigned to identify them from the rest of the system. For example, /camera_node could be a name of a node that is broadcasting camera images.

There is a rosbash tool to introspect ROS nodes. The rosnode command can be used to get information about a ROS node. Here are the usages of rosnode:

- rosnode info [node_name]: This will print the information about the node.
- rosnode kill [node_name]: This will kill a running node.
- rosnode list: This will list the running nodes.
- rosnode ping: This will check the connectivity of a node.
- rosnode machine [machine_name]: This will list the nodes running on a particular machine or a list of machines.
- rosnode cleanup: This will purge the registration of unreachable nodes.

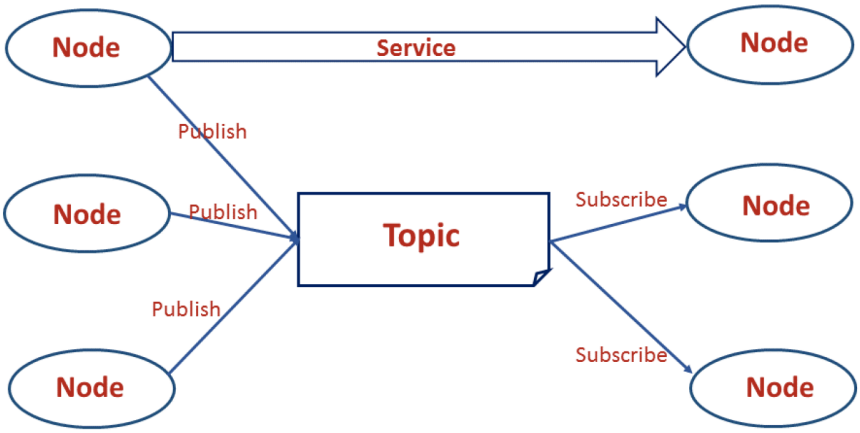

Figure A.1.: Communicating nodes.

A.2.6. Master

The ROS Master provides name registration and lookup to the rest of the nodes. Nodes will not be able to find each other, exchange messages, or invoke services without a ROS Master. In a distributed system, we should run the master on one computer, and other remote nodes can find each other by communicating with this master.

When any node starts in the ROS system, it will start looking for ROS Master and register the name of the node in it. So ROS Master has the details of all nodes currently running on the ROS system. When any details of the nodes change, it will generate a call-back and update with the latest details. These node details are useful for connecting with each node.

When a node starts publishing a topic, the node will give the details of the topic such as name and data type to ROS Master. ROS Master will check whether any other nodes are subscribed to the same topic. If any nodes are subscribed to the same topic, ROS Master will share the node details of the publisher to the subscriber node.

After getting the node details, these two nodes will interconnect using the TCPROS protocol, which is based on TCP/IP sockets. After connecting to the two nodes, ROS Master has no role in controlling them. We might be able to stop either the publisher node or the subscriber node according to our wish. If we stop any nodes, it will check with ROS Master once again.The following is a command to start ROS Master and the ROS parameter server : $ roscore

A.3. Creating a ROS Workspace & Package

The ROS packages are the basic unit of the ROS system. We can create the ROS package, build it and release it to the public. The current distribution of ROS we are using is

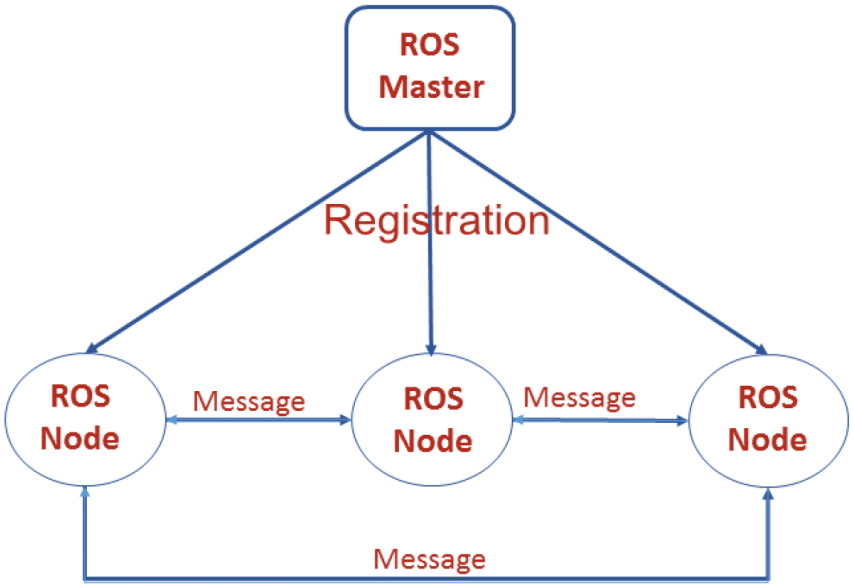

Figure A.2.: Registration and lookup the nodes.

Kinetic Kame. We are using the catkin build system to build ROS packages.

A build system is responsible for generating 'targets'(executable/libraries) from a raw source code that can be used by an end user. In older distributions, such as Electric and Fuerte, rosbuild was the build system. Because of the various flaws of rosbuild, catkin came into existence, which is basically based on CMake (Cross Platform Make).

CMake has lot of advantages such as porting the package into other operating system, such as Windows. If an OS supports CMake and Python, catkin based packages can be easily ported into it. The first requirement in creating ROS packages is to create a ROS catkin workspace.

Build a workspace folder in the home directory and create a src folder inside the workspace folder:

- $ mkdir /catkin_ws/src Switch to the source folder. The packages are created inside this package:
- $ cd /catkin_ws/src Initialize a new catkin workspace:
- $ catkin_init_workspace We can build the workspace even if there are no packages. We can use the following command to switch to the workspace folder:
- $ cd /catkin_ws The catkin_make command will build the following workspace:
- $ catkin_make

After building the empty workspace, we should set the environment of the current workspace to be visible by the ROS system. This process is called overlaying a workspace. We should add the package environment using the following command:

- $ echo "source /catkin_ws/devel/setup.bash" » /.bashrc
- $ source /.bashrc

This command will source a bash script called setup.bash inside the devel workspace folder. To set the environment in all bash sessions, we need to add a source command in the .bashrc file, which will source this script whenever a bash session starts.

This is the link of the procedure `www..wiki.ros.org/catkin/Tutorials/create_a_workspace`.

1. After setting the catkin workspace, we can create our own package that has sample nodes to demonstrate the working of ROS topics, messages, services, and actionlib.
2. The catkin_create_pkg command is used to create a ROS package in which we are going to create demos of various ROS concepts.
3. Switch to the catkin workspace src folder and create the package using the following command:

```
$ catkin\_create\_pkg [package\_name] [dependency1] [
    dependency2]
```

   Example:

```
$ catkin\_create\_pkg zaghexa\_pkg roscpp std\_msgs actionlib
    actionlib\_msgs
```

4. After creating this package, build the package without adding any nodes using the catkin_make command. This command must be executed from the catkin workspace path. The following command shows you how to build our empty ROS package:

```
$ ~/catkin_ws
$ catkin_make
```

5. After a successful build, we can start adding nodes to the src folder of this package.

Hint, The dependencies in the packages are:

- **roscpp**: This is the C++ implementation of ROS. It is a ROS client library which provides APIs to C++ developers to make ROS nodes with ROS topics, services, parameters, and so on. We are including this dependency because we are going to write a ROS C++ node. Any ROS package which uses the C++ node must add this dependency.
- **std_msgs**: This package contains basic ROS primitive data types such as integer, float, string, array, and so on. We can directly use these data types in our nodes without defining a new ROS message.

- **actionlib**: The actionlib meta-package provides interfaces to create preemptable tasks in ROS nodes. We are creating actionlib based nodes in this package. So we should include this package to build the ROS nodes.
- **actionlib_msgs**: This package contains standard message definitions needed to interact with the action server and action client.

A.4. Creating ROS nodes

A.4.1. Publisher

The first node we are going to discuss is publisher.cpp. This node will publish an integer value on a topic called /numbers. Here is the detailed explanation of the preceding code:

```
#include "ros/ros.h"
#include "std\_msgs/Int32.h"
#include <iostream>

void main()
{
   ros::init(argc, argv,"publisher");
   ros::NodeHandle node\_obj;
   ros::Publisher number\_publisher = node\_obj.advertise<std\
      _msgs::Int32>("/numbers",10);
   ros::Rate loop\_rate(10);
   while (ros::ok()) {
       std\_msgs::Int32 msg;
       msg.data = number\_count;
       ROS\_INFO("%d",msg.data);
       number\_publisher.publish(msg);
       ros::spinOnce();
       loop\_rate.sleep();
   }
}
```

The *ros/ros.h* is the main header of ROS. If we want to use the *roscpp* client APIs in our code, we should include this header. The *std_msgs/Int32.h* is the standard message definition of integer datatype. Here, we are sending an integer value through a topic. So we should need a message type for handling the integer data. *std_msgs* contains standard message definition of primitive datatypes. *std_msgs/Int32.h* contains integer message definition: *ros::init(argc, argv,"publisher");* This code will initialize a ROS node with a name. It should be noted that the ROS node should be unique. This line is mandatory for all ROS C++ nodes: *ros::NodeHandle node_obj;*

This will create a *Nodehandleobject*, which is used to communicate with the ROS system: *ros::Publisher number_publisher = node_obj.advertise<std_msgs::Int32>("/numbers",10);*

This will create a topic publisher and name the topic /numbers with a message type *std_msgs::Int32*. The second argument is the buffer size. It indicates that how many messages need to be put in a buffer before sending. It should be set to high if the data sending rate is high: *ros::Rate loop_rate(10);* This is used to set the frequency of sending data: *while (ros::ok())* This is an infinite while loop, and it quits when we press Ctrl+C. The *ros::ok()* function returns zero when there is an interrupt; this can terminate this while loop: *std_msgs::Int32 msg; msg.data = number_count;* The first line creates an integer ROS message and the second line assigns an integer value to the message. Here, data is the field name of the msg object: *ROS_INFO("%d",msg.data);* This will print the message data. This line is used to log the ROS information: *number_publisher.publish(msg);* This will publish the message to the topics /numbers: *ros::spinOnce();* This command will read and update all ROS topics, the node will not publish without a spin() or spinOnce() function. *loop_rate.sleep();* This line will provide the necessary delay to achieve a frequency of 10Hz.

A.4.2. Subscriber

After discussing the publisher node, we can discuss the subscriber node, which is subscriber .cpp. Copy the code to a new file or use the existing file.

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

/*This is a callback function that will execute whenever a data
   comes to the /numbers topic. Whenever a data reaches this topic,
    the function will call and extract the value and print it to
   the console.*/
void number_callback(const std_msgs::Int32::ConstPtr & msg)
{ ROS_INFO("Recieved [%d]",msg->data); }

void main()
{
    ros::init(argc, argv,"subscriber");
    ros::NodeHandle node_obj;

    /*This is the subscriber and here, we are giving the topic name
        needed to subscribe, buffer size, and callback function:
        > we are subscribing /numbers topic
        > we have already seen the callback function above*/
    ros::Subscriber number_subscriber = node_obj.subscribe("/
       numbers",10,number_callback);

    /* This is an infinite loop in which the node will stay:
    > This code will invoke the callbacks whenever a data reaches
       the topic.
    > The node will quit only when we press the Ctrl+C key */
    ros::spin();
}
```

This is the header needed for the subscribers.

A.5. Control Servo motor using Joystick

1. First change to the source space directory of the catkin workspace
   `$ cd ~/catkin_ws/src`
2. Now use the catkin_create_pkg script to create a new package called ‘servo1’ which depends on `std_msgs, roscpp,` and `rospy`:
   `$ catkin_create_pkg servo1 std_msgs rospy roscpp`
3. Now you need to build the packages in the catkin workspace:
   `$ catkin\_make`
4. To add the workspace to your ROS environment you need to source the generated setup file:
   `$ ~/catkin_ws/devel/setup.bash`
5. In your /home you will find the work space folder : catkin_ws inside you will find the created package which called : servo1 inside it the source folder which will include the codes & CMAKElists.txt which includes all the dependencies & the required components.
6. In folder source /src: create a new document let’s call it ’servotry.cpp’ and copy the code here inside this file & save it then close.
7. In CMakeLists.txt we will add the dependencies which is needed to run the code & for building it: please delete all comments to make the view clearer and make the code the same as this: cmake_minimum_required (VERSION 2.8.3) project(servo1)
8. Save the CMakeLists.txt then close it.
9. Important: Check that the created package: ’servo1’ folder is in the path : /catkin_ws/src if not move the folder servo1 from ’/catkin_ws’ into ’/catkin_ws/src’
10. In "servo1" folder create new folder call it "launch" & inside "launch" create new document and call it "servorun.launch" then open using text editor and insert the following code:
    As we created the code we need to build it, go to your package directory :

    ```
    $ cd catkin_ws
    $ source devel/setup.bash
    $ catkin_make
    ```

11. Now Arduino Arduino role here is to subscribe to the joystick ’the publisher’ to take the joystick readings and map it to the servo that’s connected to the Arduino

so that it gives the servo the signal to move to the specified location using given degree.

12. If you don't have Arduino IDE please check this tutorial: `www.wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup` for step by step installation once you complete the install open the Arduino IDE and jump to next step.
13. Write Arduino code:
14. Save then upload to your Arduino, don't forget to choose your board type & the serial port.

    Note: if you COULDN'T UPLOAD to Arduino & the upload stopped with the message: "*couldn't open port*", do the following:

    - from Arduino tools menu see the port address it will be like "/dev/ttyUSB0" and REMEMBER IT.
    - Then open new terminal & type:
      `$ sudo usermod -a -G dialout <TYPE YOUR USERNAME>`
    - Write your password then:
      `$ sudo chmod a+rw /dev/<TYPE YOUR PORT ADDRESS>`

      Example:

      ```
      $ sudo usermod -a -G dialout khaled
      $ sudo chmod a+rw /dev/ttyUSB0
      ```

15. After uploading the code, Open new terminal to start. don't forget to **connect the joystick**.
16. To initialize the master, type: `$ roscore`
17. Open new tab, start the communication between ROS & Arduino using serial Type :
    `$ rosrun rosserial_python serial_node.py <YOUR PORT ADDRESS>`

    Example: `$ rosrun rosserial_python serial_node.py /dev/ttyUSB0`
18. Wait to start.
19. Finally open new tab go to `catkin_ws`, then type:

    ```
    $ cd catkin_ws
    $ source devel/setup.bash
    $ roslaunch servo1 servorun.launch
    ```

Bibliography

[Byrd and de Vries, 1990] Byrd, J. and de Vries, K. A. (1990). six-legged telerobot for nuclear applications development. *Int. J. Robot*, 9:43–52. [3]

[Cousins, 2011] Cousins, S. (2011). Exponential growth of ros [ros topics]. *IEEE Robotics Automation Magazine*, 18(1):19–20. [29]

[Digia, 2017] Digia (2017). Qt project. `http://qt-project.org`. [3]

[Ding et al., 2010] Ding, X., Rovetta, A., Zhu, J. M., and Wang, Z. (2010). Locomotion analysis of hexapod robot. *INTECH Open Access Publishe*. [2]

[Dynamics, 2015a] Dynamics, B. (2015a). Dedicated to the science and art of how things move, Boston dynamics. [1, 2]

[Dynamics, 2015b] Dynamics, B. (2015b). *Boston Dynamics*. Boston dynamics. [1, 2]

[Dürr et al., 2004] Dürr, V., Schmitz, J., and Cruse, H. (2004). "behaviour-based modeling of hexapod locomotion: linking biology and technical application". *Arthropod Structure & Development*, 33:237–250. [3]

[Gurfinkel et al.,] Gurfinkel, V., Gurfinkel, E., Devjanin, E., Efremov, E., Zhicharev, D., Lensky, A., Schneider, A., and Shtilman, L. I. o. r. In six-legged walking model of vehicle with supervisory control; nauka press: Moscow, russia, 1982;. *p*, pages 98–147. [3]

[KanYoneda, 2007] KanYoneda (2007). Light weight quadruped with nine actuators. *journal of robotics & mechatronics*, 19(2). []

[Lewinger and MartinReekie, 2011] Lewinger, W. A. and MartinReekie, H. (2011). A hexapod robot modeled on the stick insect carausiusmorosus. In *the 15th international conference on advanced robotics*, Tallinn. [3]

[Lewinger and Quinn, 2010] Lewinger, W. A. and Quinn, R. D. (2010). A hexapod walks over irregular terrain using a controller adapted from an insects nervous system. In *the IEEE/RSJ international conference on intelligent robots & systems (IROS)*, pages 18–22, Taiwan. IEEE/RSJ. [3]

[Manoiu-Olaru et al., 2011] Manoiu-Olaru, S., Nitulescu, M., and Stoian, V. (2011). Hexapod robot. mathematical support for modeling and control. In *System Theory, Control, and Computing (ICSTCC), 15th International Conference on*, pages 1–6. [2]

[McGhee, 1977] McGhee, R. (1977). Control of legged locomotion systems. *In Proceedings of the*, 18:205–215. [3]

[MohdDaud and KenzoNonami, 2012] MohdDaud and KenzoNonami (2012). Autonomous walking over obstacles by means of lrf for hexapod robot comet-iv. *robotics & Mechatronics*, 24(1). [3]

[Moore and Buehler, 2001] Moore, E. Z. and Buehler, M. (2001). Stable stair climbing in a simple hexapod robot. Technical report, DTIC Document. [2]

[Okhotsimski and Platonov, 1973] Okhotsimski, D. and Platonov, A. (1973). Control algorithm of the walking climbing over obstacles. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, CA, USA, 20. Stanford. [3]

[Peternella and Salinari, 1973] Peternella, M. and Salinari, S. (1973). Simulation by digital computer of walking machine control system. In Genova, I., editor, *Proceedings of the 5th IFAC Symposium on Automatic Control in Space of the Conference*. [3]

[Saranlı, 2002] Saranlı, U. (2002). *Dynamic locomotion with a hexapod robot*. PhD thesis, The University of Michigan. [1]

[Schneider and Schmucker, 2006] Schneider, A. and Schmucker, U. (2006). Force sensing for multi-legged walking robots: Theory and experiments part 1: Overview and force sensing. In Intelligence, M. and Buchli, J., editors, *Mobile Robotics*, pages 125–174. Germany; Austria, ; Pro Literatur Verlag ARS. [3]

[Tedeschi and Carbone, 2014] Tedeschi, F. and Carbone, G. (2014). Designissues for hexapod walking robots. *Robotics*, 3(2):181–206. [1, 2]

[terrain hex-limbed extra-terrestrial explorer,] terrain hex-limbed extra-terrestrial explorer, N. A. 2009. [1, 2]