Virtual Robotic Laboratory and Learning Materials for ROSin
Education Project

Supported by ROSIN - ROS-Industrial Quality-Assured Robot Software Components.
More information: rosin-project.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 732287.

Virtual Robotic Laboratory and Learning Materials for ROSin

Education Project

Authors:

- **Foreword:** Dr. Ugur Yayan ▪ Inovasyon Muhendislik Ltd. Sti ▪ ugur.yayan@inovasyonmuhendislik.com
- **Chapter 1- 2:** Mustafa Karaca ▪ Inovasyon Muhendislik Ltd. Sti ▪ mustafa.karaca@inovasyonmuhendislik.com
- **Chapter 3:** Hakan Gencturk ▪ Inovasyon Muhendislik Ltd. Sti ▪ hakan.gencturk@inovasyonmuhendislik.com
- **Chapter 4:** Muhammed Oguz Tas ▪ Eskisehir Osmangazi University ▪ motas@ogu.edu.tr
- **Chapter 5:** Sezgin Secil ▪ Eskisehir Osmangazi University ▪ ssecil@ogu.edu.tr

Disclaimer:

Foreword

In 1920s Czech writer Karel Čapek has derived the concept of robots from Czech word called “robota” which means serves to human. This concept has pioneered the ideas of easying human life with use of robots. Especially cross-country trade, wars, production costs and efforts to reach space have been effective in the emergence of many technological innovations in the field of robotics. Various mobile and industrial robots developed provide many benefits in hospitals, hotels and industrial areas today. Especially mobile and industrial robots, which have a significant impact on the market since the 1980s, will play a key role in the smart building ecosystem that will be developed in the future with the industry 4.0 movement. Today, efforts are being made to make these robots smart. Many software platforms have been developed today to understand the complexity of making robotic systems smart and to realize the newly developed systems to certain standards. Robot operating system (ROS) has reached high popularity among these platforms. One of the main criteria of ROS platform’s success is that it has open source codes and is constantly supported by a large community. Various resources are provided for learning the ROS platform, which is relatively difficult to understand, but it is important to learn the latest practices in developing technology and developing open source architecture. Especially in our country, it is very difficult to find a resource that deals with Linux, Gazebo and ROS comprehensively and has today's applications. In this book, where we bring together the most appropriate theory and practice together with our experienced engineer staff, many important concepts related to ROS are discussed.

With the path followed in this book, it is aimed that every reader who put precious effort will become ROS compatible software manufacturer. In this book, where we bring together the most appropriate theory and practice together with our experienced engineer staff, many important concepts related to ROS are discussed. With the path followed in this book, it is aimed that every reader who has worked has become a ROS compatible software manufacturer. With the interactive topics described in the book, the reader does not just pass by reading the topics, It is aimed that the reader will become a permanent ROS compatible software provider by learning the principles in the book with the examples in the book and learning the principles it contains by practicing without memorizing it. Since it is thought that a programming education will not be successful without practicing and touching the keyboard in this process, the book was built on practicing the reader directly with

the applications, and it was ensured that each reader gain knowledge and practice with coding. With this meticulously prepared book, the development of new engineers that will contribute to future smart platforms is the greatest dream of the authors and it is aimed that the readers will benefit from this book.

Contents

Figures

1. Introduction

Dear reader,

This book is aimed to explain the Robot Operating System (ROS) software, which provides great benefit and convenience in the programming of robotic systems, and to continue the subjects by learning the topics and concepts related to ROS with examples. Each of the titles presented aims to ensure that the reader has build good infrastructure about ROS and become ROS-compliant software manufacturer without having difficulty.

With the active introduction of robots since 2000s, the value of robotic systems, which will have important features in the future, with today's smart technologies, is constantly increasing. It is important to create new values for this issue, which is of great importance today and in the future. Also, generating values for ROS compatible software can be relatively difficult for newcomers. Today, when it is difficult to find a resource covering the latest techniques related to ROS in the constantly renewing robotic field, with this textbook resource that we offer, the education will start with the history and philosophy of Unix, which is the basis of the Linux operating system. The initial chapters that continue with the introduction and development of the Linux operating system will continue with the establishment of Ubuntu, learning the basic commands. Afterwards, important applications about Python programming will be made and a detailed and applied narrative about the basic python knowledge learned and the ROS interlayer will be presented. Afterwards, important information about Gazebo, which can be used in accordance with ROS and has advanced features, is explained and detailed and exemplary studies about MoveIt are presented. Each chapter presented in the book is constructed with the information added on the previous one and it is important to follow the chapters in the book in the order given, since no chapter is seen separately. It is thought that this book will be a learning tool for every reader who spends effort and at the end of the training, every reader that puts efforts on these chapters are expected to become a ROS Compatible software manufacturer.

2. Linux History and Ubuntu

2.1. History

2.1.1. Unix

The existence of the Unix operating system is based on MULTICS, that is, the multiplexer operating and compensation system. The MULTICS project started in the mid-1960s with the cooperation of General Electric, the Massachusetts Institute of Technology and Bell labs, and in 1969 the Bell Lab was withdrawn from the project. One of the people working on this project was Ken Thompson. He saw the great potential of MULTICS. However, MULTICS was very complex and Ken Thompshon felt that the same work would be done more simply. In 1969, he wrote the first version of UNIX called UNICS. It represented the UNICS Uniplex Information and Compuing system, ie the one-sided operating and calculation system. Although the operating system changed over time, the name was shortened to the same pattern, Unix.

Ken Thompson worked with Dennis Ritchie, who wrote the first C compiler. In 1973, they rewrote the Unix kernel at C. The next year, a version of Unix, known as the Fifth Edition, was first licensed to universities. The Seventh Edition, released in 1978, became a split point for two different Unix development lines and was divided into two branches, SRV4 (system V) and BSD. This operating system, which has different versions, was licensed to different institutions and commercialized. Unix unilateral calculation can be called the philosophical main source of success.

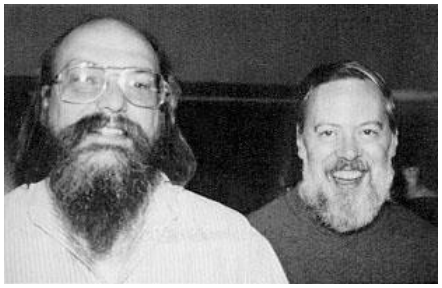

Figure 1 Ken Thompson and Dennis Ritchie

Unix pholosophy:

- Worse is better.
- Simple, easy to use, minimalist, reusable software.
- 4 rules: Simple, accurate, consistent, complementary.

- Each program does a job, but it does it in the best way.
- Instead of adding new features to old programs, write new ones from scratch.
- Let the output of one program be the input of another program.

Everything is a file in the Unix philosophy..

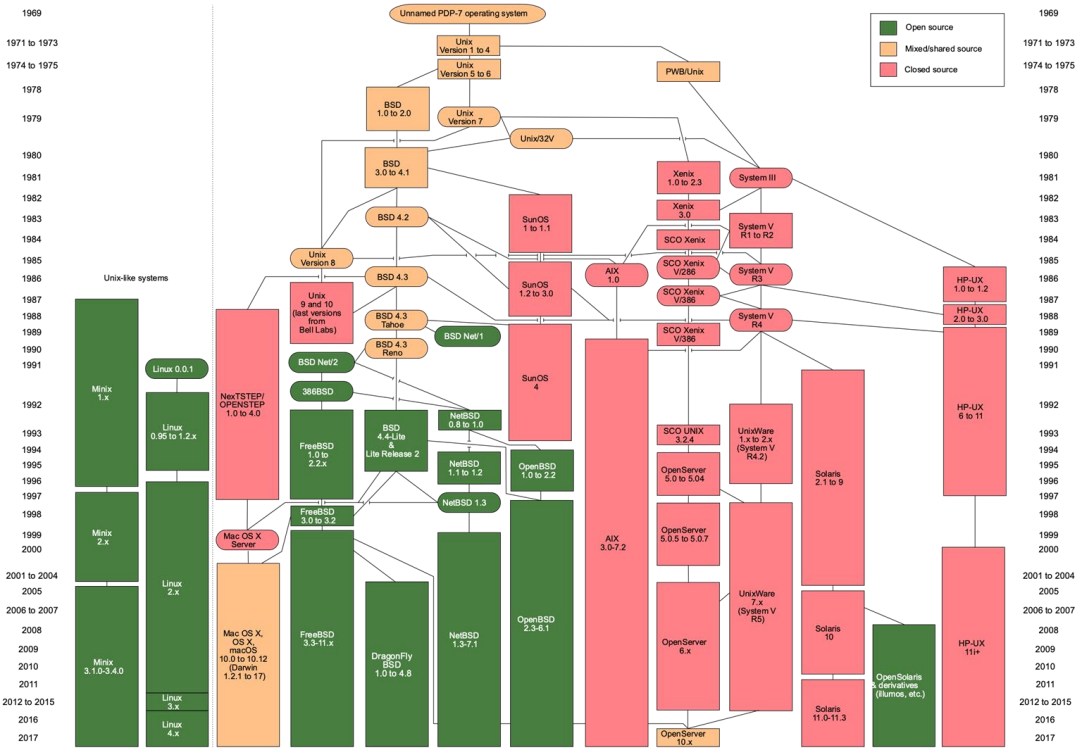

Figure 2 Development of Unix and Unix-like systems

2.1.2. GNU

In 1983, Richard Stallman overcame the goal of creating an operating system that he called GNU, which is similar to the UNIX operating system but will be free and open source. He knew that the most important requirement of this difficult target was the development of the core that forms the center of an operating system. Since the development of this kernel called GNU Herd is progressing very slowly, Linux / GNU operating systems and distributions started to emerge with the combination of the Linux kernel and GNU particles published by Finnish Linus Torvalds, a university student in the 1990s. Therefore, it is understood that Linux

operating system is actually composed of Linux kernel and other components under the GNU. This structure is called Linux / GNU. Since 2012, it has been completed with the official introduction of the Linux kernel to the GNU project.

2.1.3. Linux

University student Linus Torvalds found Linux core and version 1.0 was released in 1994. This operating system consisting of Linux / GNU generally consists of 6 particles. These are:

- Linux kernel
- GNU tools and libraries,
- Documentation,
- Window system (The most used and traditional X Window System is used in Wayland today),
- My Desktop Environment,
- Package managers

Linux has an important role today with its open source license, the freedom to run programs for any purpose, the study mechanism of the program, the freedom to distribute the developed codes to others.

2.1.4. Distributions

Linux offers different versions to suit all types of users. These versions are called distribution / distributions. Today there are more than 500 active distributions. It treats the desktop of each distribution differently. Some prefer very modern user interfaces (such as GNOME and Elementary OS's Pantheon), while others stick to a more traditional desktop environment (using openSUSE, KDE).

2.1.5. Package managers

One of the Linux distributors was said to be a package manager. Package management is compiled and stored in a repository (repo) for different CPU and hardware configurations, due to the difficulty of compiling each project with open source distribution and compiling separately on each machine. Users can

download and use it. The Linux kernel supports many configurations. These are examples of important architectures such as x86-32, x86-64, ARM-32 and ARM-64. The most important part that separates Linux distributions is the package management system. Debian and Ubuntu based distributions use the apt system and .deb packages, Arch based ones use the pacman manager, Fedora and RedHat derivatives .rpm package structure.

Distributions	Package System
Debian	apt (dpkg, .deb)
Red Hat, CentOS, Fedora	yum (.rpm)
SUSE	YaST
Arch	pacman
Pardus	PiSi

2.1.6. Ubuntu

Ubuntu, Canonical Ltd. is a Unix-based operating system written by a private company with approximately 450 employees developed by the company. There are 2 versions per year and these are 0.4 and .10 versions. LTS versions of these versions receive “Long Term Support”, ie long-term 5-year development support. Today 16.04. and 18.04 versions have LTS feature.

2.1.7. Shell

User-operating system communication is provided through a program called “shell”. These can be based on Shell Graphic user interface (GUI) or Command-line interface (CLI). Shell provides access to the services of the operating system, and the cmd.exe program known in the Windows operating system is an example. Shells used historically in Linux can be said as csh, tcsh, ksh, zsh, ash. The most popular shell used today is GNU Bash. If commands entered at the command line in GNU Bash are written one after another in a file, the Bash interpreter can execute this script.

2.2. Ubuntu Installation

All available ubuntu versions are available at http://releases.ubuntu.com/. The downloaded Ubuntu .iso file should be written to Usb memory or CD. Unetbootin or Rufus programs can be used for printing.

Figure 3 Available Releases

In the second stage of the Ubuntu installation, the Rufus program is downloaded and an external USB disk is selected from the device section (1). Afterwards, the boot type will be selected (2). After selecting the boot type, the directory of the .iso file is displayed (3). After this process, the start button is pressed and when the memory is ready, the close button can be pressed.

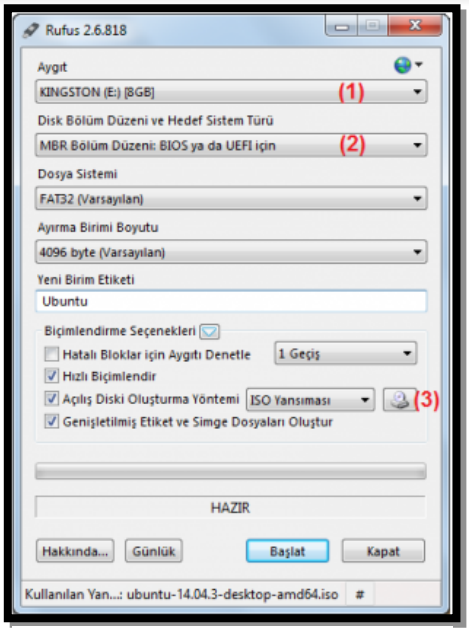

Figure 4 Rufus Program

Insert the USB memory prepared while booting the computer and press F2 or F12 to select the USB memory to be used for Ubuntu installation from the boot settings. There is an option that you can try without installing Ubuntu on the screen that appears:

- With the "Try Ubuntu" option, you can try Ubuntu version and exit without any registration.
- Ubuntu installation can be started with the "Install Ubuntu" option or during the trial.

An example screenshot of the options is given in figure 5.

Figure 5 Installation Screen

You can start the installation of Ubuntu by pressing Install Ubuntu. The following options can be selected for installing updates that have been installed during installation and for installing side technologies. These sections are not compulsory. The user can also make updates after the installation process is finished, if he wishes. The screenshot that you may encounter in this process is shown in figure 6.

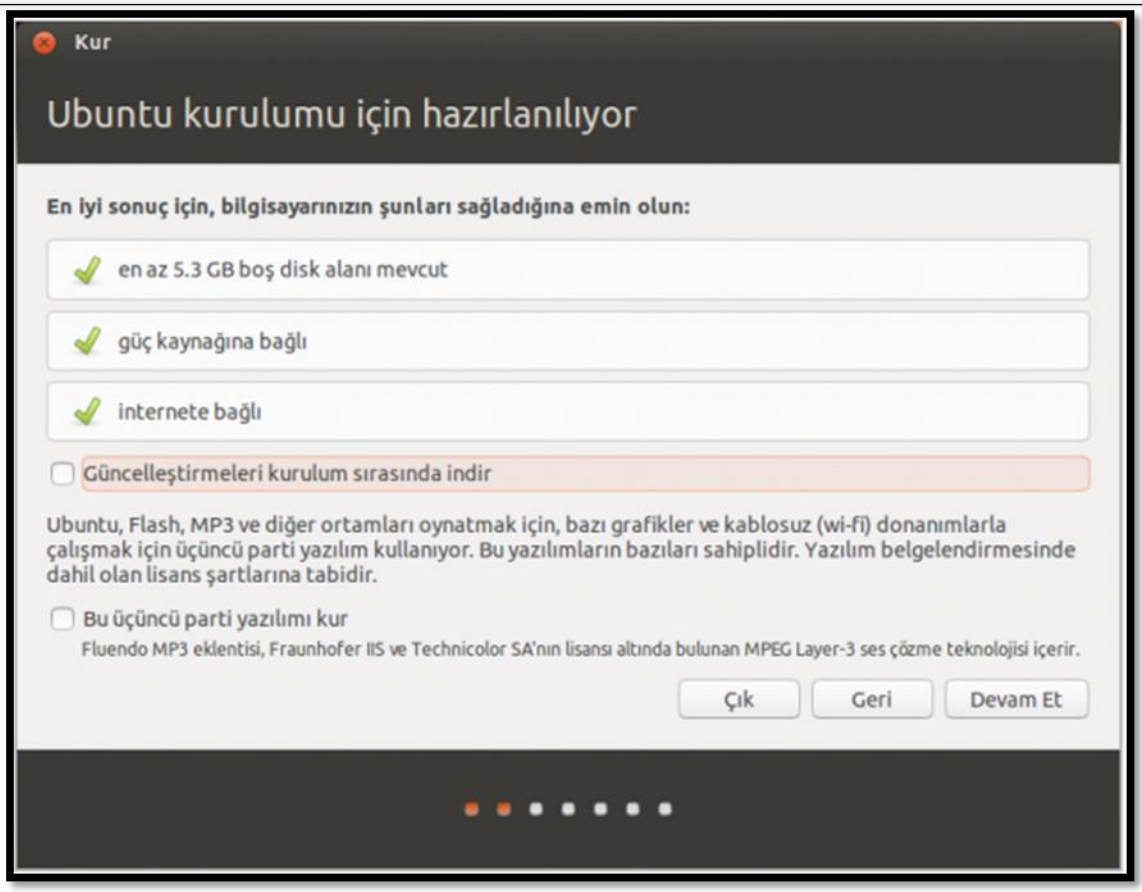

Figure 6 Preparation for Installation

As in Figure 7, in the Setup Type window you will see during installation:

- Replace the Windows operating system with Ubuntu: This option deletes Windows and the documents on your computer and installs Ubuntu instead.
- Encrypt the new Ubuntu installation for security: With this option, we can control the access and intervention for non-ourselves by adding a password to the installation.
- Use LVM with new Ubuntu installation: With this option, we install on the disk that is configured to be expandable. This does not require a special disc. If we make this selection, our disk is configured in accordance with the feature of increasing the size of our disk without damaging the data in the future.
- Another thing: This option allows you to configure the disc to your own needs. Using this option is somewhat complicated and requires experience.

- If no operating system is installed on your computer, the first option will be Erase Disk and install Ubuntu.

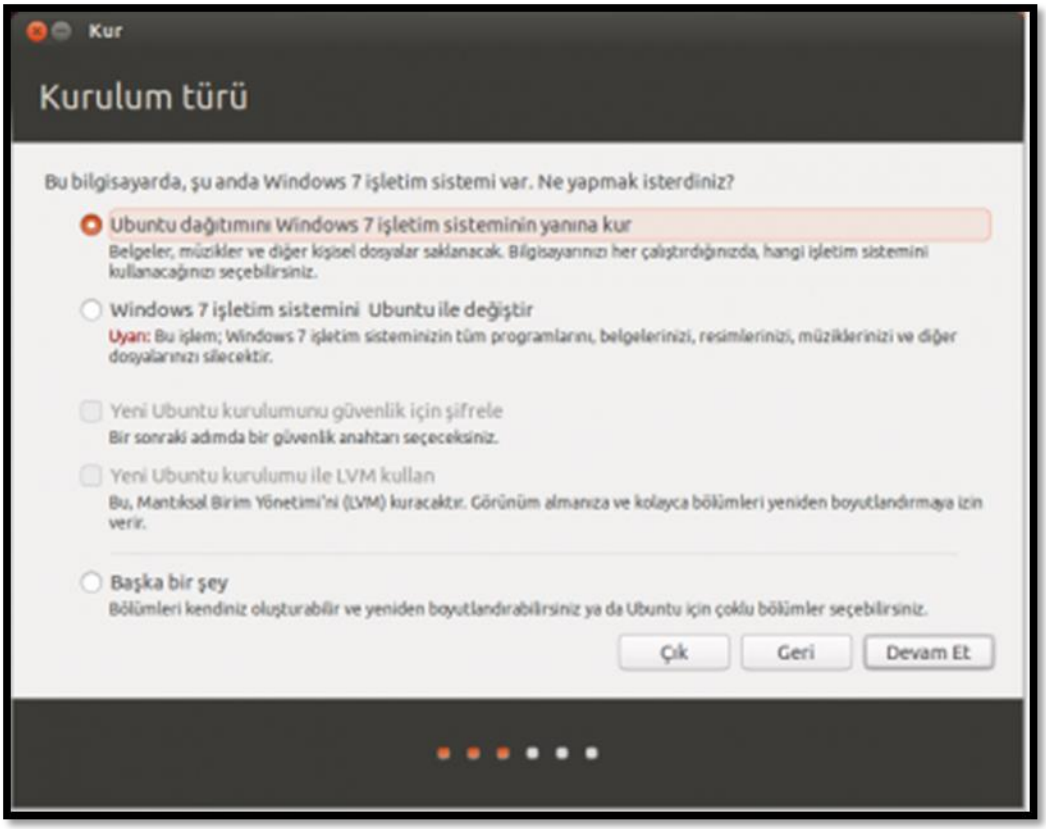

Figure 7 Installation Options

After this process, "Where are you?" The screen shown in figure 8 asking the question will be encountered. Here you can select the time zone of your country. selectable options for Istanbul Turkey.

Figure 8 Regional Time Settings

In the next step of the installation, you will see the "Keyboard layout" screen. Usually, we do not need to take any action since our keyboard layout will be determined automatically under normal conditions. If it cannot be determined automatically, the 'Detect Keyboard Layout' button in the lower left part can also be used to detect the keyboard. If the desired keys are pressed here, your keyboard layout will be detected. In addition, after selecting a keyboard language, you can check the correctness of the language, that is, compatibility with your keyboard by entering some Turkish characters from the keyboard in the box here.

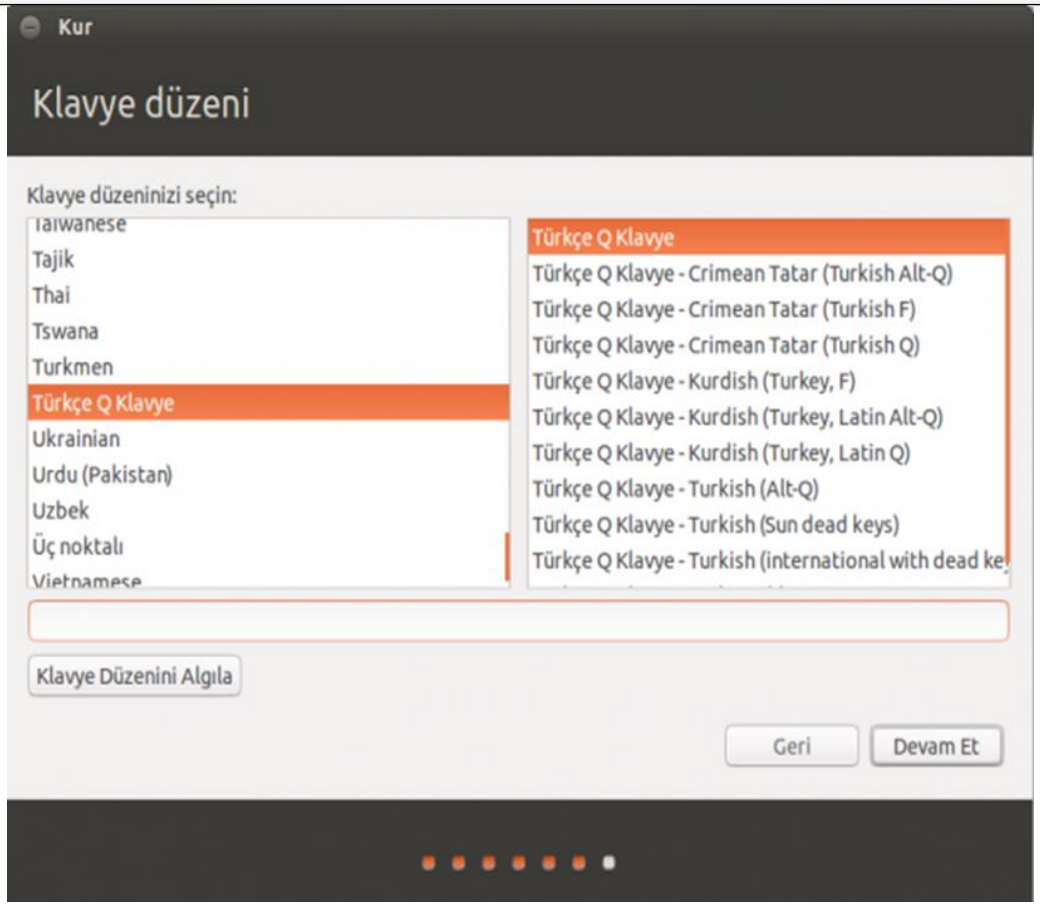

Figure 9 Keyboard Layout Screen

In the next step, "Who are you?" section will come, username and password will be determined. At this stage, we enter the name of our computer and the name of our user account. If the password we set is at least 6 characters long and includes characters such as letters, numbers,? * - etc., Ubuntu setup will like our password and say 'Good Password'.

- **Auto login:** You can select this option if you want to log in directly without entering your user password every time you open Ubuntu. Even if you do not select this option, you can also enable automatic user login through Settings after installing Ubuntu.
- **Require password to login:** It is the opposite of the option above. This option is checked by default.

- **Encrypt my home directory:** By checking this option, the user home directory under the folder named / Home can be encrypted for access outside of us by encrypting our home directory.

Then, when we use the 'Continue' button, the "Install" screen, which comes up with the "Installing System" message, starts to complete the installation.

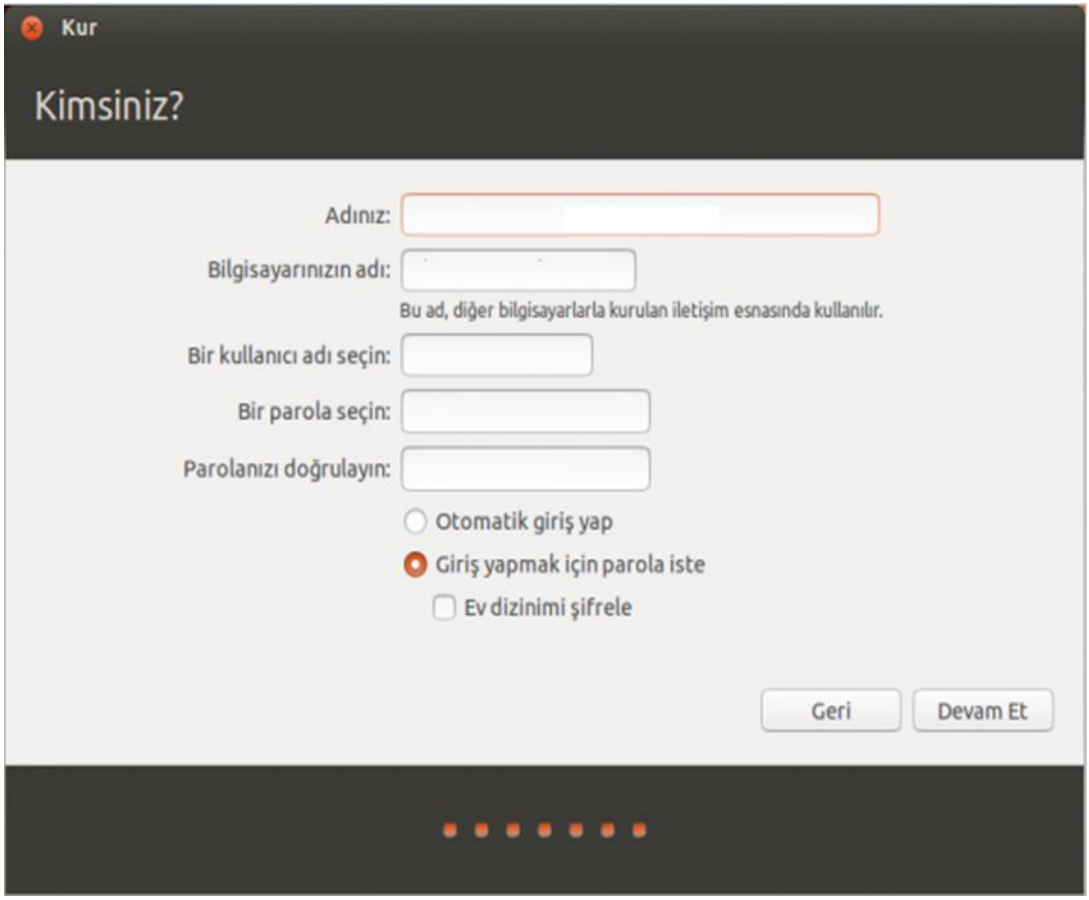

Figure 10 Login Screen

After this process, wait for the installation to finish. When the installation is complete, we receive the message "Installation Complete". After saying "Restart Now" we can now start using Ubuntu.

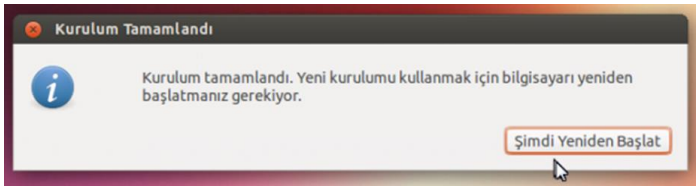

Figure 11 Installation completion Message

2.3. Ubuntu Basics

2.3.1. Prompt (Command Prompt)

Command prompt A program that takes commands entered by users and sends them to the operating system for processing. It can show the results and output of commands on itself. That is, it creates a bridge between the operating system and the user. As an example of the command prompt, the command prompt belonging to the user "yonetici" is given in figure 13.

Figure 12 Sample Command Prompt

```
yonetici@yoneticipc:~$ whoami
yonetici
yonetici@yoneticipc:~$ hostname
yoneticipc
yonetici@yoneticipc:~$ lscpu
yonetici@yoneticipc:~$ uname
Linux
yonetici@yoneticipc:~$ uname –r
5.0.0-29-generic
```

Figure 13 Sample Application

whoami: Gives your username you typed while logging into the system.

Hostname: Gives the host name of the machine.

Lscpu: Shows CPU information that reports the number of CPU, thread, core, socket, and Malfunctioning Memory Access (NUMA) nodes.

Uname: is the screen where system information is shown, in addition to uname to print certain information:

-a, --all

- If not known omit Print all information except -p and -i in the following order.

-s, --kernel-name

- Kernel name is written.

-r, --kernel-release

- Prints kernel relase

-v, --kernel-version

- Prints kernel version

-m, --machine

- print machine hardware name

-o, --operating-system

- print operating system

--help

- Shows and exits help section

--version

- the version prints out and exits.

```
$ mkdir siirler
$ cd siirler
$ pwd
/home/yonetici/siirler
```

Figure 14 Sample Application

mkdir: The mkdir command is used to create a new directory.

Cd: It is the command used to go to a directory which means “change directory”. It is case sensitive and the name of the folder must be typed in full.

Pwd: This command can be used if you want to know which file is in it.

As an example, it is seen that the user creates the poems folder with “mkdir” in figure 14 and enters this folder with the command “cd” and controls it with “pwd”.

2.3.2. Downloading Files

Wget: command used to download files. A sample download address is given in figure 15 for an example application.

```
$ wget http://188.132.181.140/otuzbes.txt
$ wget http://188.132.181.140/handuvar.txt
$ wget http://188.132.181.140/sessiz.txt
```

Figure 15 Sample wget Application

2.3.3. Listing Files

Ls: Enables listing of files in the folder. By adding -1 command, it can be shown on separate lines and detailed information can be accessed with the -l command. Example application is shown in figure 16.

```
$ ls
handuvar.txt  otuzbes.txt  sessiz.txt

$ ls -1
handuvar.txt
otuzbes.txt
sessiz.txt

$ ls –l
-rw-r--r-- 1 yonetici yonetici 6662 Sep 23 01:46
handuvar.txt
-rw-r--r-- 1 yonetici yonetici 1163 Sep 21 02:09
otuzbes.txt
-rw-r--r-- 1 yonetici yonetici 528 Sep 23 01:49
sessiz.txt
```

Figure 16 Sample ls Application

2.3.4. Displaying Files

More: Used to display a text file at a time. 1 page is displayed at a time. Press the space to move to the next page..

Less: Although it is almost the same as more, it has navigation that enables the transition to the up and down page which is not possible in the "more" command.

Cat(Concatenation): can be used to combine multiple files and print the result on the screen.

An example application for the use of these commands can be seen in figure 17.

```
$ more otuzbes.txt
$ less handuvar.txt
$ cat sessiz.txt
$ cat otuzbes.txt handuvar.txt sessiz.txt
$ cat *
```

Figure 17 Sample Application

2.3.5. Head and Tale

Head: displays the first ten lines of a file. It shows the sections after the desired line with the -n command.

Tale: Displays the last part of the file.

An example application for their use is shown in figure 18.

```
$ head otuzbes.txt
$ tale handuvar.txt
$ head –n 5 sessiz.txt
```

Figure 18 Sample Application

2.3.6. Standard Flows

Computer programming includes standard input, standard output and output indicating standard error for pre-determined inputs and outputs. “0” output for standard input, “1” output for standard output and “2” output for standard error are specified. This system is shown in figure 19.

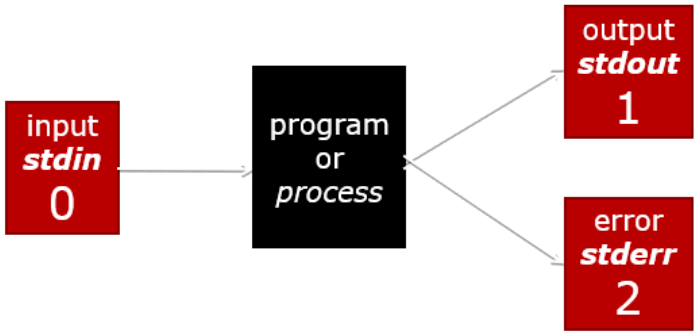

Figure 19 Standard Flow System

2.3.7. Redirecting

>: used to provide commanded output.

```
$ cat otuzbes.txt handuvar.txt >
ikisiir.txt
$ more ikisiir.txt
```

Figure 20 Sample Application

<: used to provide command input.

```
$ cat < handuvar.txt
```

Figure 21 Sample Application

|: used to use the result of one command in another command. How to use and work in Figure 22 and Figure 23 are shown.

```
$ echo "Benim Siirim" | cat sessiz.txt -
```

Figure 22 Sample Application

Figure 23 working process

2.3.8. Regular Expressions(regex)

Regular expressions represent special characters used to match complex patterns, assist in the search for data.

Symbol	Expression
.	Replaces any character
^	Matches string start
$	Matches string end
*	Matches zero or more times of the previous character
\	Represents special characters
()	Represents regular expressions in a group
?	matches only one character

As an example, figure 24 and figure 25 can be seen.

Figure 24 Sample Application

```
$ grep "^B" sessiz.txt
Bicare gonuller! Ne giden son gemidir bu!
Bilmez ki giden sevgililer donmeyecekler.
Bir cok gidenin her biri memnun ki yerinden,
Bir cok seneler gecti; donen yok seferinden.
```

```
$ grep '!$' handuvar.txt
    Yalniz arabacinin dudaginda bir islik!
    Artik bahtin aciktir, uzun etme, arkadas!
    Basucumda gordugum su satirlarla yandim!
    Ey Marasli Seyhoglu, evliyalar adagi!
    Bahtina lanet olsun asmadinsa bu dagi!
    Donmeyen yolculara aglayan yasli yollar!
```

Figure 25 Sample Application

2.4. Linux File System

Unix can be thought of as an upside down tree. At the top of this tree is "root", that is the root of the tree. All the directories shown in Figure 26 are based on root.

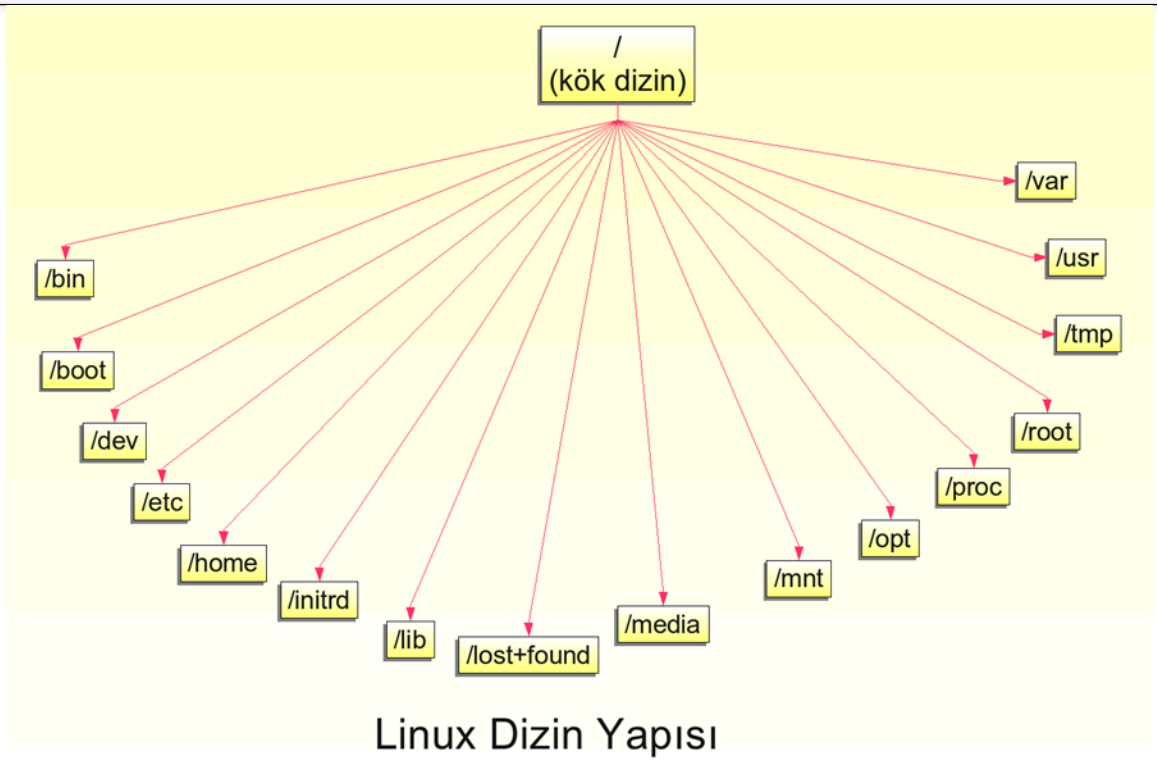

Figure 26 Linux Root Directory Structure

Basic Files below the root are:

directory	Descriprion
/bin	Includes must-have scripts
/boot	Contains files necessary for startup
/dev	Contains hardware files
/etc	Contains system settings
/lib	Contains library files and kernel modules
/media	It is the folder where removable devices (CD-Rom, Flash memory etc ...) are added to the system.
/mnt	Used to temporarily add a file system.
/opt	For installing extra programs
/sbin	Keeps executable files related to the system administrator.

/srv	Related to the services provided by the system
/tmp	For holding temporary files
/usr	A secondary hierarchy
/var	Stores variable data

2.4.1. Sudo

Sudo: "super user do" which means allow non-root users to run normally require superuser permissions commands.

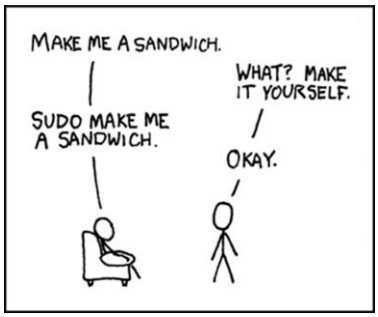

Figure 27 What is Sudo?

2.4.2. users

Root: defined as superuser. It is a super user-defined root name in the system. This user has the right to make any changes to the system. The sudo command is used to eliminate the security risks it brings and to run some processes at the root level first.

Useradd or **adduser** commands can be used to add a new user to the system. Using the **adduser** command is recommended because it works at a high level. (See figure 28 for an example.)

```
$ man adduser
```

Figure 28 Sample adduser application

2.4.3. File Permissions

Each file has an owner and a group.

Rights are defined for the following users for a file:

- User (u)
- Group (g)
- Others (o)

Identifiable rights are as follows:

- Read – 4
- Write – 2
- Execute - 1

For example, when "4: 2: 1" is given, the expression (r: w: x) is defined, that is, the user has the right to read, write and operate. An example application is given in figure 29.

```
$ ls -l
-rw-r--r-- 1 yonetici yonetici 6662 Sep 23 01:46
handuvar.txt
-rw-r--r-- 1 yonetici yonetici 1163 Sep 21 02:09
otuzbes.txt
-rw-r--r-- 1 yonetici yonetici 528 Sep 23 01:49
sessiz.txt
$ chmod go-w handuvar.txt
$ chmod u+x otuzbes.txt
```

Figure 29 Sample Permission Application

2.4.4. Processes

ps: Refers to Processes. Lists processes running on Linux system.

ps -aux | grep <işlem_ismi>: Returns the specific process or processes running on Linux.

kill -9 <action_ID>: Allows killing the process running on Linux and having the ID with the command above. The first running process (number 1) is init.

2.4.5. Package Installation

wget '<download_url>': Downloads the file specified on the internet to the folder on the computer.

sudo apt-get install <package_name>: Searches for the repository and installs it on the computer.

sudo apt-get remove <package_name>: Searches for and deletes the package from the computer.

sudo apt-get update: retrieves the information of the repository stored in the sources.list file to the computer.

sudo apt-get upgrade: Updates the packages on the computer.

apt-cache search <package_name>: Searches and fetches the related package in repository.

sudo chmod <permission_type> <folder_or_file_name>: Gives file permissions to the relevant folder or document. Using 777 as the permission type means Read-Write-Execute.

2.4.6. Services

sudo service <service_name> start: Starts the service running within Linux.

sudo service <service_name> stop: Terminates the service running within Linux.

sudo service <service_name> restart: Restarts the service running under Linux

Startable services

- FTP
- Web server (NGINX, Apache)
- Databases (MySql, PostgreSQL)
- SSHD (remote connection)

2.4.7. Text Editors

Text editors can be used to write code, edit text files such as configuration files, create user instruction files, and more. On Linux, text editor is of two types, the graphical user interface (GUI) and command line text editors (CLI).

CLI based:

- pico, nano
- vi, vim
- Emacs

GUI based:

- pico, nano
- vi, vim
- Emacs

2.4.8. Linux Terminal

It provides an interface where users can type scripts and print text. When you SSH your Linux server, the program you run on your local computer and write commands is a terminal. Some shortcuts for the terminal:

Terminal: Ctrl + Alt + T
Copy: Ctrl + Shift + C
Paste: Ctrl + Shift + V
Terminate: Ctrl + C

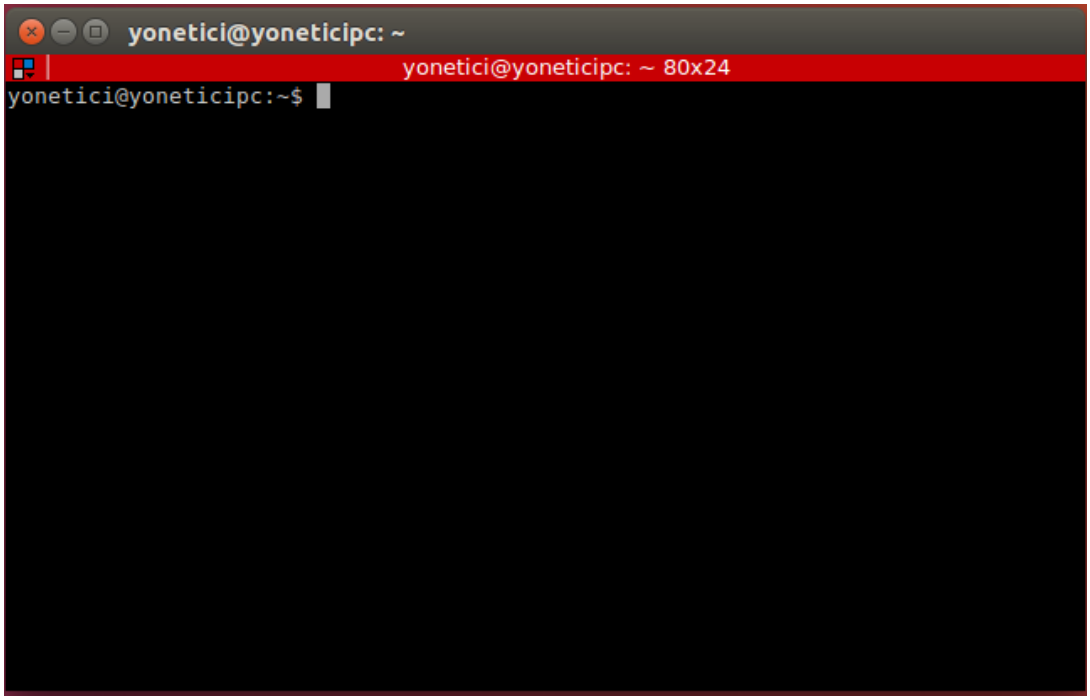

Figure 30 Terminal Window

In the sample command window shown in Figure 30, "yonetici @ yoneticipc:" is written. While the **yonetici** represents the username, **yoneticipc** represents the computer name.

3. Introduction to Python Programming

If you want to run phyton code, you can go directly to the directory where the file is located from the terminal, "python filename.py" can be written to run the code, but in order to be productive with ROS, the programming language must be known. The basic terms are given in figure 31.

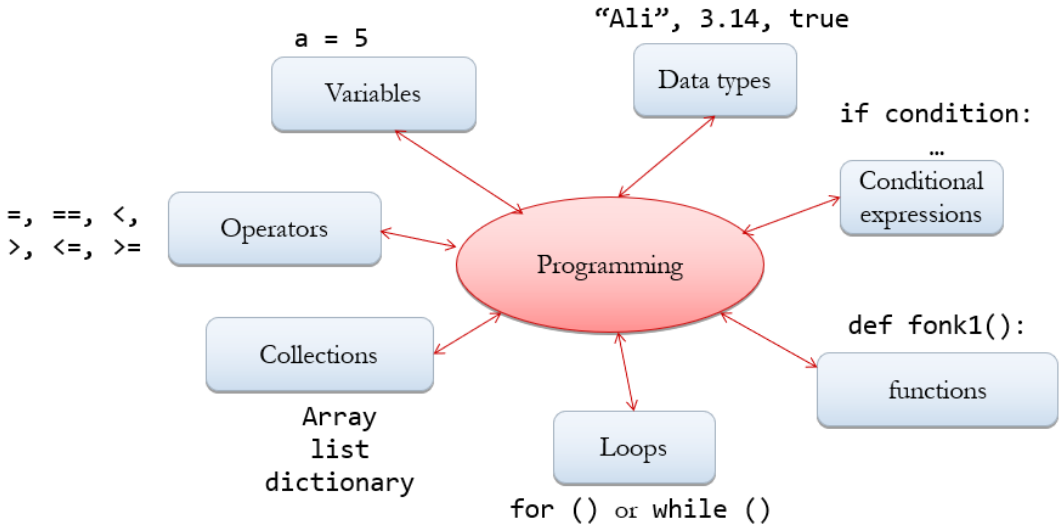

Figure 31 Basic Programming Terms

3.1.1. Variables

Example of variables:

- a = 5
- b = 'Deneme'
- c, d = 10, 'Deneme2'

3.1.2. Data types

Data types include integers, decimal numbers (floating), texts (string) and true / false (bool). These are:

Integer: 0,1,2,3,…-1,-2,-3….
Float:1.234, 0.94, 1009.75, -100.32 …
Sting: ”Ali”, “Beşiktaş”, “Türkiye” ….
Bool: True, False

exampled.

3.1.3. Conditional Expressions

We use three statements to indicate condition conditions:

- İf
- elif
- else

Conditional expressions are used below.

```
if koşul_1:
    sonuç_1
elif koşul_2:
    sonuç_2
elif koşul_3:
    sonuç_3
else:
    sonuç_4
```

3.1.4. Functions

If a simple function is defined:

```
def basitFonksiyon():
    print('Deneme')
```

If the defined function is called as follows:

```
basitFonksiyon()
```

“Trial” will appear on the terminal screen. If another addition is defined:

```
def toplama(a, b):
    sonuc = a + b
    return sonuc
```

Calling this function will be:

```
sonuc = toplama(5, 10)
```

3.1.5. Loops

Loops in most other programming languages pass through a numerical order where the programmer can specify the start and end points, while loops are performed by repeating any sequential data type in Python for sample loops and their usage patterns:

```
while koşul:
    …

for i in range(3, 20):
    print(i)

for harf in metin:
    print(harf)
```

Can be given. If another application is to be made:

```
parola = input("parola giriniz: ")

if not parola:
    pass
```

In this application, if the user passes the password blank, it means that they cannot do anything thanks to the pass command. For the break command:

```
while True:
    sayi = input("Bir sayı girin: ")
    if sayi == "iptal":
            break
```

If the user writes a cancellation, the loop is exited. For the continue command:

```
for val in "string":
    if val == "i":
        continue
```

The continue command is used to jump without exiting the loop if it satisfies the condition.

3.1.6. Collections

Liste(list): if a list defined:

```
liste = [50, 60, 'Merhaba', True, 99]
```

In this list:

```
print(liste[2])              # 'Merhaba'
```

```
print(liste[1:3])          # 60, 'Merhaba'
print(len(liste))          # 5
```

will be outputs**. liste.append("…")** adds new element **liste.sort()** sorts the list.

Sözlük(dictionary): If a sample dictionary is written:

```
sozluk = {
     'isim': 'Ayse',
     'sehir': 'Eskisehir',
     'yas': 22
    }
```

To print the name of ayşe:

```
print(sozluk['isim'])      # Ayse
```

Command can be used.

3.1.7. Operators

There are basically 7 operators in Python. These are:

Operator	Description
<	smaller
<=	smaller than or equal
>	bigger
>=	greater than or equal
=	append

==	equal
!=	not equal
is	object equality
is not	negative object equality

3.1.8. Finding Prime Numbers

```
sayi = int(input("Sayi Giriniz:"))
sayac = 0

for i in range(2, (sayi + 1)):
    kontrol = True
    for j in range(2, i):
        if(i % j == 0):
            kontrol = False
            break
    if kontrol:
        print(i)
        sayac += 1

print("Toplam ", sayac, " adet asal sayı vardır.")
```

As an output, a sample output is shown in Figure 32.

```
Sayi Giriniz:25
2
3
5
7
11
13
17
19
23
Toplam  9  adet asal sayı vardır.
```

Figure 32 Sample Application for Finding Prime Numbers

3.1.9. Plot Application

To download the Matplotlib library:

- Pip install matplotlib

To download Pip:

- sudo apt install python-pip

ommands should be written.

```
import matplotlib.pyplot as plt  # importing matplotlib
library

x = [1, 2, 3]                    # x axis values
y = [2, 4, 1]                    # y axis values

plt.plot(x, y, "--r", label = "Line 1")       # to draw plot
with dots
```

```
plt.xlabel('X - Ekseni')                # x axis name
plt.ylabel('Y - Ekseni')                # y axis name

plt.title('Plot Ornegi')                # adding header to plot
plt.show()                      # showing plot function
```

In this application, an output is expected as given in figure 33..

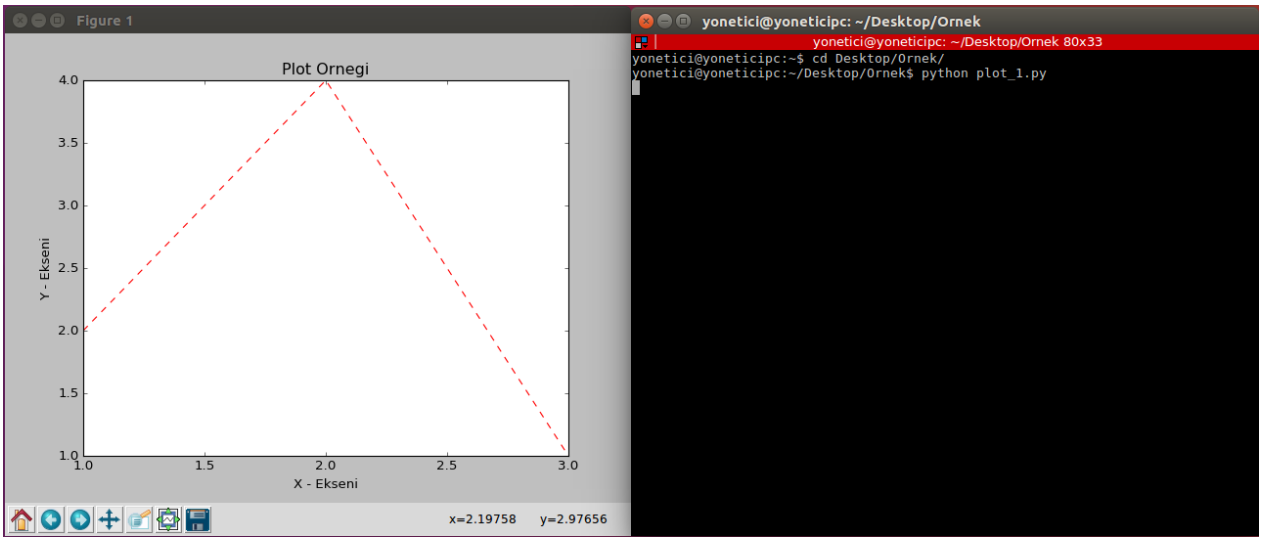

Figure 33 Sample Plot Application

3.1.10. Dual Plot Application

If code down below is added:

```
import matplotlib.pyplot as plt

x1 = [1,2,3]
y1 = [2,4,1]
plt.plot(x1, y1, "--r", label = "line 1")

x2 = [1,2,3]
```

```
y2 = [4,1,3]
plt.plot(x2, y2, "-.g", label = "line 2")

plt.xlabel('X - Ekseni')
plt.ylabel('Y - Ekseni')
plt.title('Ikili Plot Ornegi')

plt.legend()
plt.show()
```

In this application, an output is expected as given in Figure 34..

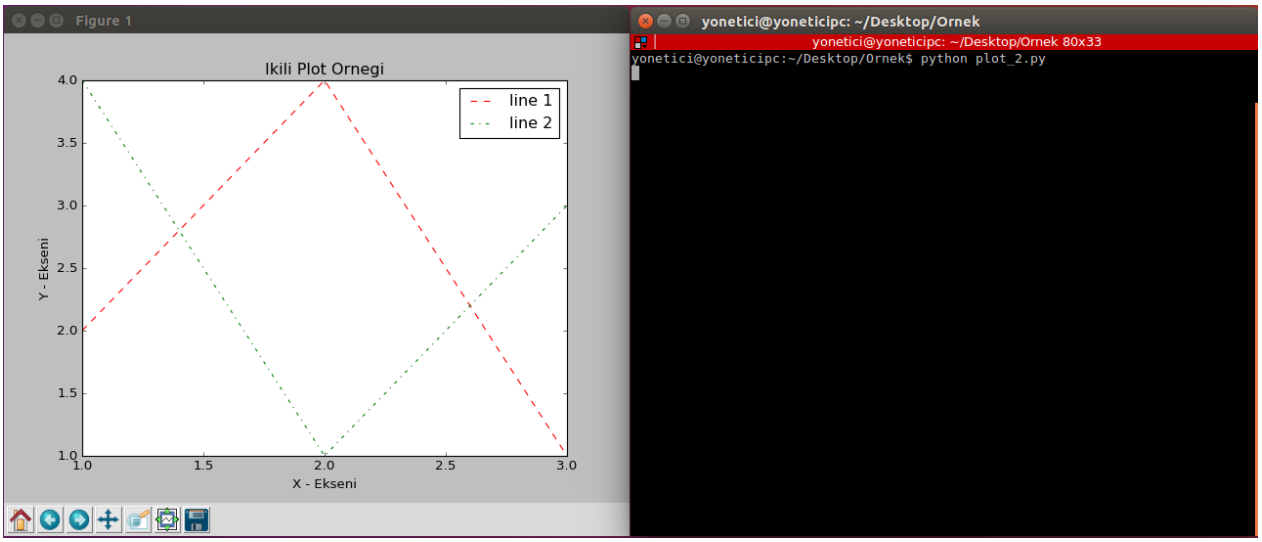

Figure 34 Sample Dual Plot Application

4. ROS Applications

4.1. What is ROS - what is not?

ROS stands for Robot Operating System is an open source meta operating system. It functions as a robotic interfacer with hardware abstraction, low-level device control, realization of commonly used functionality, messaging between

processes, and packet management. The communication topology of ROS is based on the network structure from terminal to terminal. Thanks to this, data losses due to slowness in communication of computers connected with a heterogeneous network are prevented. ROS consists of four main concepts: node, message, subject and service. The node is where processes are performed. Communication between these nodes takes place via messages. The messages flow over the topics. Topics communicate in the publisher / subscriber structure. Therefore, the subjects provide asynchronous communication. Services are used instead of subjects for synchronous communication.

ROS:

- It is not a programming language.
- It is not a library.
- is not an operating system.
- It is not an integrated development environment.
- It is a meta operating system for open source robots that run services and various macros.

4.2. Why should we use ROS?

ROS has many useful features. Examples of these are:

- Language independent structure (C++, Python, Lisp, Java, Lua)
- Modular run, parameters, messages and services allow instant intervention
- Systematic data transfer with Node/topic
- Driver support for many sensor, motor and robot platform
- Open source
- Algorithms, libraries and packages for mapping, localization and detection
- Active community
- Rapid testing
- Harware abstraction
- Visualizors

4.3. ROS supported Sensors and Companies Using ROS

Ros offers many possibilities to use sensors. There are different sensors supported by ROS. These are:

- single size range finders (1D range finder)
- 2D range finder
- 3D Sensors (rangefinder and RGB-D cameras)
- Voice / Speech Recognition sensors
- Environmental sensors (eg ultrasonic wind sensor)
- Force / Torque / Touch Sensors
- Motion capture
- Exposure Prediction (GPS / IMU)
- Power source
- RFID reader
- Radar sensors

Figure 35 Boston Dynamics Atlas Robot

Today, many companies use ROS interfaces in robot development. Some of these companies are:

- ABB Robotics
- Clearpath Robotics
- Doosan Robotics
- Fanuc
- Fetch Robotics
- iRobot
- Sony (Aibo robot dog)

Figure 36 EvaRobot

All the robots produced by Innovation Engineering operate in ROS compliance. Among these robots are the intelligent wheelchair (ATEKS) training and research robot (EvaRobot) and the robots it produces for endustrial use.

4.4. ROS Installation

ROS setup is generally recommended with debian packages. An alternative to Debian packages is to install using the ROS source code, but this is not recommended. ROS Kinetic Kame distribution supports Xenial (Ubuntu 16.04) and Willy (Ubuntu 15.10) versions.

Figure 37 Platform Selection Window

Figure 38 Version Selection Window

From the address given for installation, version 37 and Figure 38 can be installed by selecting the ROS kinetic Kame.
http://wiki.ros.org/kinetic/Installation

If you want to install from the source code, you can use the link below:
http://wiki.ros.org/kinetic/Installation/Source
Things to do at this stage are given below.

Setting up the computer to accept the software from settings.ros.org:

```
sudo sh -c 'echo "deb
http://packages.ros.org/ros/ubuntu   $(lsb_release -
sc) main" > /etc/apt/sources.list.d/ros-latest.list‘
```

Setting up keys:

```
sudo apt-key adv --keyserver
'hkp://keyserver.ubuntu.com:80' --recv-   key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Making sure the Debian package is up to date:

```
sudo apt-get update
```

Installing the full version(ROS,rqt,rviz,robot-generic libraries, 2D/3D simulators ...):

```
sudo apt-get install ros-kinetic-desktop-full
```

Starting and updating rosdep:

```
sudo rosdep init
rosdep update
```

Environment Setup:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
    source ~/.bashrc
```

Installing dependencies for compiler packages:

```
sudo apt install python-rosinstall python-rosinstall-
generator python-wstool build-essential
```

4.5. Linux Basic Codes

Basic codes about Linux terminal:

- Openning new terminal: Ctrl + Alt + T
- yonetici@yoneticipc:
  - yonetici: user name
  - yoneticipc: computer name
- Copy: Ctrl + Shift + C
- Paste: Ctrl + Shift + V
- Terminate Process: Ctrl + C

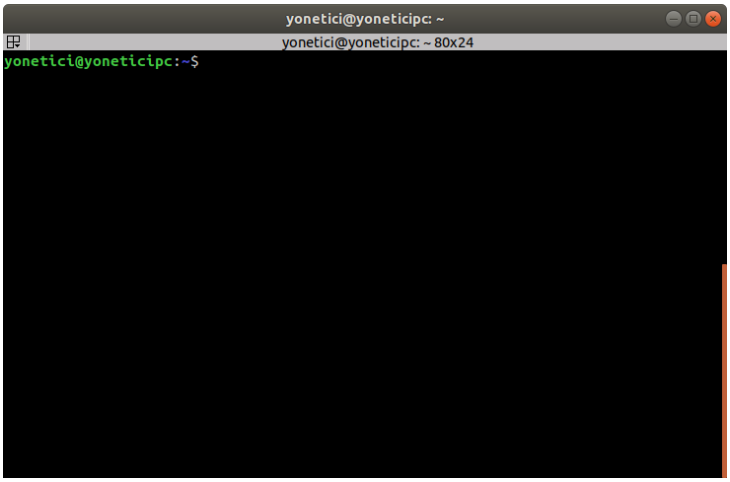

Figure 39 Linux Terminal Window

n addition, the Linux basic commands mentioned in the previous topic that may be required when operating on ROS are given below:

- **cd :** means «Change Directory». Enters the folder on the specified path.
- **ls :** means «List». Lists the files and documents in the entered folder.
- **mkdir <directory_name>:** means «Make Directory». Creates directory in given path.
- **mkdir -p <directory_path>/<dierctory_name>:** It realizes the folder creation process all the way. If there are non-existent folders, it creates nested folders.
- **rm <file_to_be_deleted>:** means «Remove». Removes specified directory..
- **rm -rf < file_to_be_deleted >:** means «Recursive Remove». deletes multiple files.

- **mv < file_to_be _moved> < directory_to_be _moved>:** Performs file transfer. Rename can also be done using the mv command.
- **cp < file_to_be _coppied> < folder_to_be _coppied>:** used for copying.
- **wget '<download_url>' :** It downloads the file specified on the internet to the folder on the computer.
- **sudo apt-get install <package_name>:** Searches and installs the package on repository.
- **sudo apt-get remove <package_name>:** Searches for and deletes the package from the computer.
- **sudo apt-get update:** Retrieves the information of the repository stored in the sources.list file to the computer.
- **sudo apt-get upgrade:** Updates the packages on the computer.
- **apt-cache search <package_name>:** Searches and fetches the relevant package in repositories.
- **sudo chmod <permision_type> <file/directory_name>:** Gives file permissions to the relevant folder or document. Using 777 as the permission type means Read-Write-Execute.
- **sudo su:** allows doing operations as a super user.
- **sudo service <service_name> start :** Starts service running on Linux.
- **sudo service < service_name> stop :** Terminates service running on Linux.
- **sudo service < service_name> restart :** Restarts the service running on Linux.
- **history:** Shows code history in terminal.
- **clear:** Deletes codes in the terminal.
- **ps :** means «Processes». Lists processes running on Linux system.
- **ps -aux | grep <process>:** Returns the specific process or processes running on Linux.
- **kill -9 <process_ID> :** Allows killing process running on Linux and having ID with command above.
- **udo lsusb :** Lists USB devices registered and running on Linux system.
- **cat :** Print the contents of a file on the terminal screen.
- **pwd :** Suppresses the path of a file to the terminal screen.
- **echo <variable> :** Allows printing of global variables and variables defined later on the Linux terminal to the screen.

- **<editör_name> <file> :** Allows editing a file on Linux.
- **setxkbmap tr :** maps the keyboard to use the layout determined by the options specified on the command line(tr= Turkish).
- **g++ hello_world.cpp -o hello_world :** Compiles C / C ++ files via GNU C Compiler.
- **./hello_world :** Executable file execution

4.6. ROS Architecture

ROS architecture is divided into 3 parts:

- **Filesystem Level:** Contains a group concept to explain how ROS is internally created, the folder structure and the minimum number of files that should work.
- **Computation Graph Level:** Contains concepts that explain how ROS uses communication between processes and systems.
- **Community Level:** Includes a set of tools and concepts for sharing information, algorithms and code among developers.

4.7. File System Level

Ros package system consists of 5 parts:

- Packages
- Metapackages
- Package Manifests
- Message types
- Service types

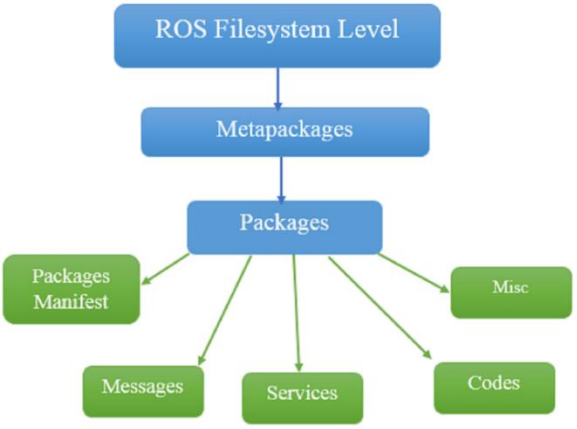

Figure 40 ROS File System

4.7.1. Packages

A package can contain processes (nodes), ROS-linked libraries, datasets, configuration files, or anything else useful.The purpose of creating the packages is

to divide the codes into small pieces and make them reusable.Packages are the main units that keep the software organized in ROS.Packages usually consist of typical files and folders::

- **include / package_name /:** This directory contains the header files of the libraries we need.
- **msg /:** If a message type other than the standard message types will be created, it must be created in this folder.
- **Scripts /:** Codes written in script languages such as Python should be placed in this folder.
- **src /:** Where the source files of the programs are located.
- **srv /:** This is where the service files are located.
- **CMakeLists.txt:** It is a CMake compilation (built) file containing the orders given to the compiler and many more building information.
- **package.xml:** The package file of the packages

ROS has tools that can help us create, edit, and work with packages:

- **rospack:** To find and learn about packages.
- **roscreate-pkg:** Creates a new package.
- **rosmake:** It is used to compile packages.
- **rosdep:** It is used to load dependencies of packages.
- **catkin_create_pkg:** Creates a new package.

ROS has a package called rosbash that allows us to move between packages and folders and files of packages. Some commands supported in the rosbash tool:

- **rospack:** Enables browsing between ROS directories.
- **rosed:** Enables editing files.
- **roscp:** Enables copying files from another package.
- **rosrun:** Enables executable files to run.
- **rosls:** It is used to list the files in the package.

The package must contain the CMakeLists.txt file. This file tells catkin how and where to load the codes.The CMakeLists.txt file should follow the format below, otherwise packages cannot be created correctly.

```
Satır numaralandırmayı açkapa

  1  # Get the information about this package's buildtime dependencies
  2  find_package(catkin REQUIRED
  3    COMPONENTS message_generation std_msgs sensor_msgs)
  4
  5  # Declare the message files to be built
  6  add_message_files(FILES
  7    MyMessage1.msg
  8    MyMessage2.msg
  9  )
 10
 11  # Declare the service files to be built
 12  add_service_files(FILES
 13    MyService.srv
 14  )
 15
 16  # Actually generate the language-specific message and service files
 17  generate_messages(DEPENDENCIES std_msgs sensor_msgs)
 18
 19  # Declare that this catkin package's runtime dependencies
 20  catkin_package(
 21   CATKIN_DEPENDS message_runtime std_msgs sensor_msgs
 22  )
 23
 24  # define executable using MyMessage1 etc.
 25  add_executable(message_program src/main.cpp)
 26  add_dependencies(message_program ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_T
ARGETS})
 27
 28  # define executable not using any messages/services provided by this package
 29  add_executable(does_not_use_local_messages_program src/main.cpp)
 30  add_dependencies(does_not_use_local_messages_program ${catkin_EXPORTED_TARGETS})
```

Figure 41 CMakeList.txt

- **Required CMake Version** (cmake_minimum_required)
- **Package Name** (project())
- **Find other CMake/Catkin packages needed for build** (find_package())
- **Enable Python module support** (catkin_python_setup())
- **Message/Service/Action Generators** (add_message_files(), add_service_files(), add_action_files())
- **Invoke message/service/action generation** (generate_messages())
- **Specify package build info export** (catkin_package())
- **Libraries/Executables to build** (add_library()/add_executable()/target_link_libraries())
- **Tests to build** (catkin_add_gtest())
- **Install rules** (install())

4.7.2. Metapackages

Meta packages are used to run packages organized (simply group multiple packets).Meta packages are special packages in ROS that contain only the package.xml file.

example: robot metapackage includes **packages**: [control_msgs, diagnostics, executive_smach, filters, geometry, joint_state_publisher, kdl_parser, kdl_parser_py, robot_state_publisher, urdf, urdf_parser_plugin, xacro]

```
sudo apt-get install ros-$distro-robot
```

```
sudo apt-get install ros-$distro-actionlib ros-$distro-angles ros-$distro-bond_core ros-$dist
ro-catkin ros-$distro-class_loader ros-$distro-cmake_modules ros-$distro-common_msgs ros-$dis
tro-console_bridge ros-$distro-control_msgs ros-$distro-diagnostics ros-$distro-dynamic_recon
figure ros-$distro-executive_smach ros-$distro-filters ros-$distro-gencpp ros-$distro-geneus
 ros-$distro-genlisp ros-$distro-genmsg ros-$distro-gennodejs ros-$distro-genpy ros-$distro-g
eometry ros-$distro-message_generation ros-$distro-message_runtime ros-$distro-nodelet_core r
os-$distro-pluginlib ros-$distro-robot_model ros-$distro-robot_state_publisher ros-$distro-ro
s ros-$distro-ros_comm ros-$distro-rosbag_migration_rule ros-$distro-rosconsole_bridge ros-$d
istro-roscpp_core ros-$distro-rosgraph_msgs ros-$distro-roslisp ros-$distro-rospack ros-$dist
ro-std_msgs ros-$distro-std_srvs ros-$distro-xacro
```

Figure 42 ROS Robot Metapackage Distributions

4.7.3. Package Manifests

Package notifications (**package.xml**) is a file that contains other information about a package: its name, version, description, license information, dependencies, and exported packages. The reason why this file was created is to facilitate package loading and distribution..

Detailed information for the sections in the package is given in figure 42.

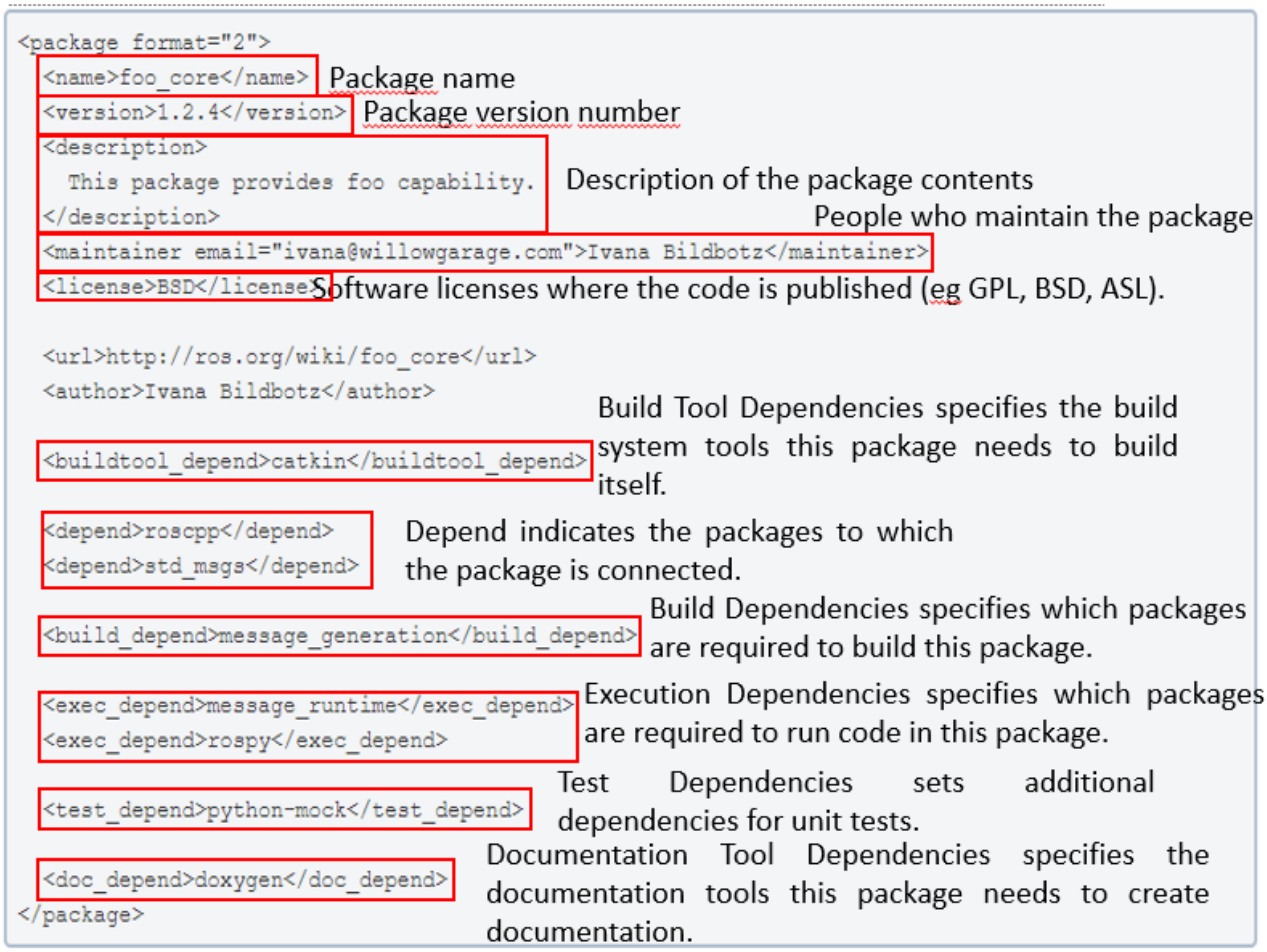

Figure 43 Package.xml Parts and Definitons

4.7.4. Message types

The message file must be in the **msg /** folder and have the extension **.msg** (my_package / msg / MyMessageType.msg).

Figure 44 Pose Message

4.7.5. Service types

The service file must be in the **srv /** folder and have the extension **.srv** (my_package / srv / MyServiceType.srv).

turtlesim/Spawn Service

File: turtlesim/Spawn.srv

Raw Message Definition

```
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be created and returned if this is empty
---
string name
```

Compact Message Definition

```
float32 x
float32 y
float32 theta
string name
---
string name
```

Figure 45 TurtleSim Spawn Service

4.8. Computation Graph Level

The ROS process graphic level consists of 7 sections:

- Nodes
- Parameter Service
- Messages
- Topics
- Services
- Bags
- Master

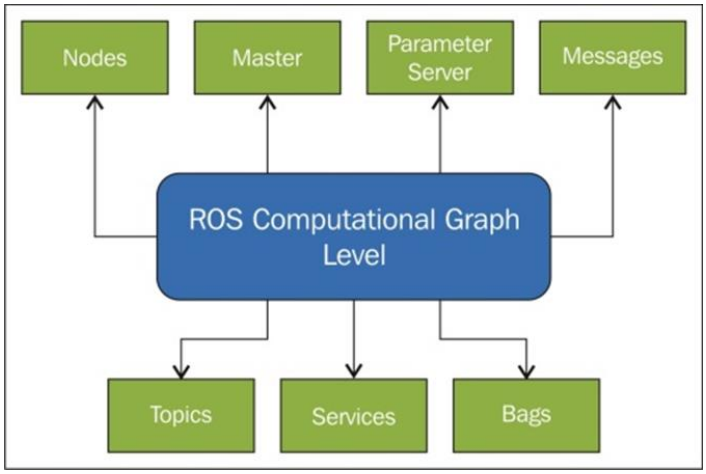

Figure 46 ROS Computational Graph Level

4.8.1. Nodes

Nodes are computations.A node can be written using different libraries such as roscpp for C ++ and rospy for Python.Using nodes in ROS gives us fault tolerance and simplifies the system and functions, separating the code and functions.

A node must have a unique name in the system.A strong feature of ROS nodes is the ability to change parameters (node name, subject name, etc.) when starting the node. With the modification process, the node can be configured without recompiling the code, so it can be easily adapted to different scenarios.

- Example of changing the topic name in the node:
  - rosrun book_tutorials tutorialX topic1:=/level1/topic1
- Example of changing parameters in node:
  - Rosrun book_tutorials tutorialX _param: = 9.0

ROS has another node type called nodelets. These special nodes are designed to run multiple nodes in a single operation. With this, nodes can communicate more efficiently without overloading the network. Nodelets are especially useful for camera systems and 3D sensors where the volume of data transferred is very high.

Not: : Instead of having a large node that does everything in the system, it is more efficient to have many nodes that provide only one functionality.

ROS has the rosnode tool to process nodes and provide information. Some commands supported in the rosnode tool:

- **rosnode info node_name:** Prints information about the node.
- **rosnode kill node_name:** Terminates a running node.
- **rosnode list:** Lists active nodes.
- **rosnode machine hostname:** Lists the nodes running on a particular machine.
- **rosnode ping node_name:** Tests the connection to the node.

4.8.2. Parameter Service

With parameters, it is possible to configure running nodes or change the operating parameters of a node.ROS has the rosparam tool to work with Parameter Server. Some commands supported in the rosparam tool:

- **rosparam list:** Lists all parameters on the server.
- **rosparam get parameter:** Gets the value of a parameter.
- **rosparam set parameter:** Sets the value of a parameter.
- **rosparam delete parameter:** Deletes a parameter.
- **rosparam dump file:** Saves the parameter server in a file.
- **rosparam load file:** Loads a file (with its parameters) on the parameter server.

4.8.3. Messages

Nodes communicate with each other through messages. A message contains data that provides information to other nodes.A message consists of two parts, type and name. We can create our own message type.

In ROS, you can find a lot of standard types to use in messages, as shown in the following table list:

Primitive type	Serialization	C++	Python
bool (1)	unsigned 8-bit int	uint8_t(2)	bool
int8	signed 8-bit int	int8_t	int
uint8	unsigned 8-bit int	uint8_t	int(3)
int16	signed 16-bit int	int16_t	int
uint16	unsigned 16-bit int	uint16_t	int
int32	signed 32-bit int	int32_t	int
uint32	unsigned 32-bit int	uint32_t	int
int64	signed 64-bit int	int64_t	long
uint64	unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string (4)	std::string	string
time	secs/nsecs signed 32-bit ints	ros::Time	rospy.Time

Figure 47 ROS Message Standards

Headers are a special type in ROS. The timeline has the numbering system and sequence number that let us know who the messages are coming from.

Figure 48 ROS Header Message

ROS has a tool called rosmsg that allows us to see the message definition and the source file where the message type is specified. Some commands supported in the rosmsg tool:

- **rosmsg show:** Displays the fields of this message.
- **rosmsg list:** Lists all posts.
- **rosmsg package:** Lists all messages in the particular package.
- **rosmsg packages:** Lists all packages with messages.
- **rosmsg users:** Searches for code files using the message type.

4.8.4. Topics

The messages are routed through a transport system with broadcast / subscribe semantics. A node sends a message by posting a specific subject. The subject is a name used to describe the content of the message. A node that deals with a particular type of data will subscribe to the appropriate topic. It is important that the subject names are unique to avoid confusion. You can have multiple concurrent publishers and subscribers for a single topic, and a single node can broadcast and / or subscribe to multiple topics.

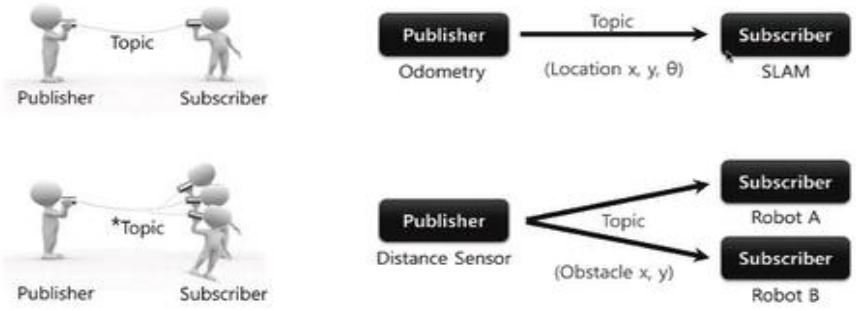

Figure 49 Publisher / Subscriber

ROS has a tool to work on topics called *rostopic*. Some commands supported in the *rostopic* tool:

- **rostopic bw/topic:** Shows the bandwidth used by the topic.
- **rostopic echo/topic:** Print messages on the screen.
- **rostopic find message_type:** Find topics by type.
- **rostopic hz/topic:** Shows the publish rate of the topic.
- **rostopic info/topic:** Prints information about the topic, such as message type, publishers, and subscribers.
- **rostopic list:** Prints information on active topics.
- **rostopic pub/topic type args:** Publishes relevant data. It enables us to create and publish data directly from the command line on the topic we want.
- **rostopic type/topic:** Prints the subject type, that is, the type of message it posts.

4.8.5. Services

The broadcasting / subscribing model is a very flexible communication paradigm, but many-to-many, one-way transportation is not generally suitable for request / response interactions desired in a distributed system. The request / response is done through services defined by a double message structure: one for request and the other for response. The provider provides a service under a name and uses a service by sending a customer request message and waiting for the answer. It has rossrv and rosservice command line tools to work with ROS services. Some commands supported in these tools:

- **rosservice call/service args:** Calls the service with the given arguments.
- **rosservice find msg-type:** Finds service by service type.
- **rosservice info/service:** Prints information about the service.
- **rosservice list:** Lists active services.
- **rosservice type/servis:** Prints service type.
-

4.8.6. Bags

The bag is a file created by ROS, created in .bag format to record all information of all messages, subjects, services and other information and then play it back. Bags are an important mechanism for storing data, such as sensor data, which can be difficult to collect but is required to develop and test algorithms. Tools that can be used in ROS to use bag files:

- **rosbag:** Used to record, play and perform the requested data.
- **rqt_bag:** It is used to visualize the data in graphic environment.

4.8.7. Master

The part of the nodes in ROS that facilitates communication with each other is called ROS master. ROS Master provides search to the rest of the Trading Chart. Without a master, the nodes cannot find each other, exchange messages or call for service. Before operating any ROS node, we must start the ROS Master and ROS parameter server. We can start the ROS Master and ROS parameter server using a single command called roscore. In the texts we encounter:

- In the first part, a log file is created inside the ~ / .ros / log folder to collect the logs from the ROS nodes. This file can be used for debugging purposes.
- In the second part, a ROS initialization file called roscore is launched. This section shows the address of the ROS parameter server in the port.
- In the third section, parameters such as rosdistro and rosversion are displayed.

- In the fourth section, it is seen that the rosmaster node was started using ROS_MASTER_URI, which we previously defined as the environment variable.
- In the fifth chapter, it is seen that the rosout node has started.

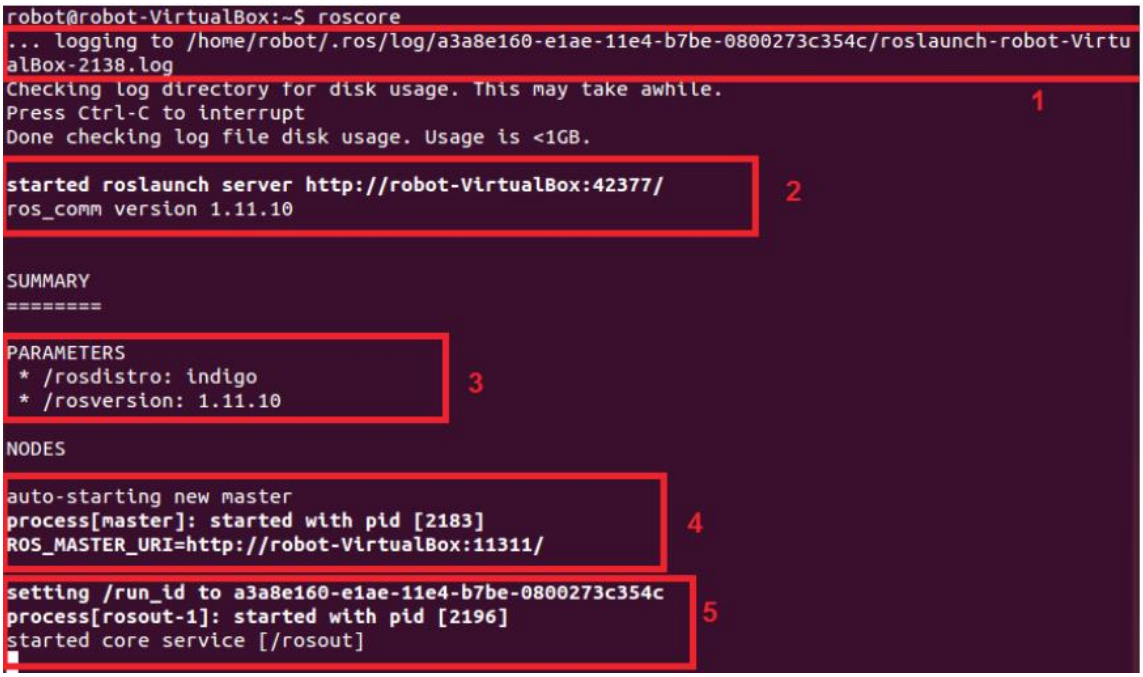

Figure 50 RosMaster

4.9. Community Level

- **Distributions:** ROS Distributions are collections of version stacks that you can load.
- **Repositories:** ROS offers a code repository where different organizations can develop and publish their own robot software components.
- **ROS Wiki:** The main forum that documents information about ROS.
- **Mail list:** The Ros-users mailing list is the primary communication channel, a forum that asks questions about the ROS software as well as new updates to ROS.
- **ROS Answers:** It is a question and answer site to answer your questions about ROS.

- **Blog:** http://www.ros.org/news , provides regular updates, including photos and videos.

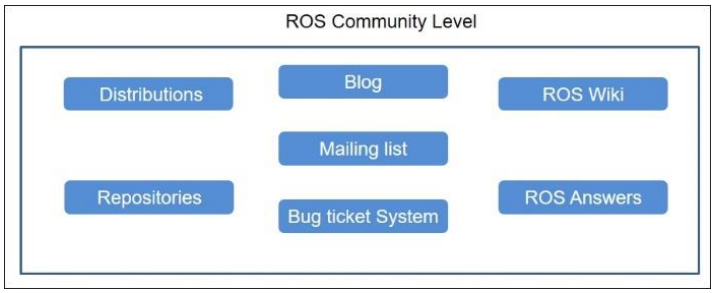

Figure 51 Community Level

4.10. Publisher-Subscriber and Service Client Structure

Publish-Subscribe: This communication model requires that the message be broadcast without the publisher explicitly specifying the recipients or having the knowledge of the intended recipients. The subscriber records the relevant ones from the published messages.

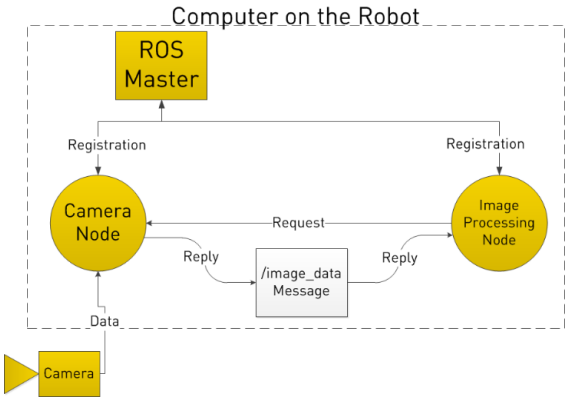

Figure 52 Publish-Subscribe example

Service-Client: It is a communication model that provides one-time communication and the customer sends the request and the server returns a response. Used when the robot is asked to perform a special task (for example, from point A to point B).

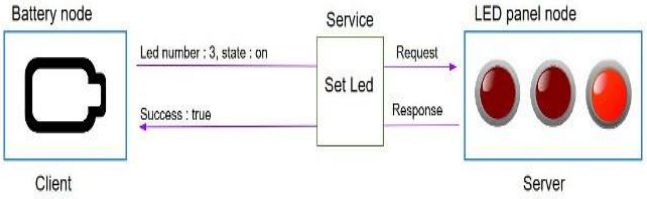

Figure 53 Service-Client example

4.11. ROS Tools

ROS has several GUI and command line tools to inspect and debug messages. Some of those:

- **rviz:** One of the 3D visualizers available in ROS to visualize 2D and 3D values from ROS topics and parameters.
- **rqt_plot:** A tool for drawing scalar values in the form of ROS topics.
- **rqt_graph:** Visualizes the connection graph between ROS nodes.

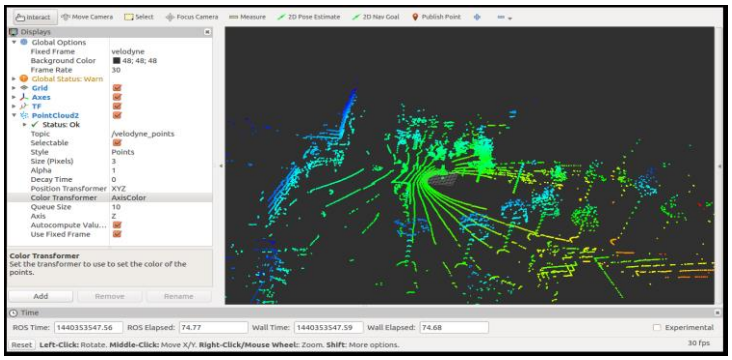

Figure 55 Rviz

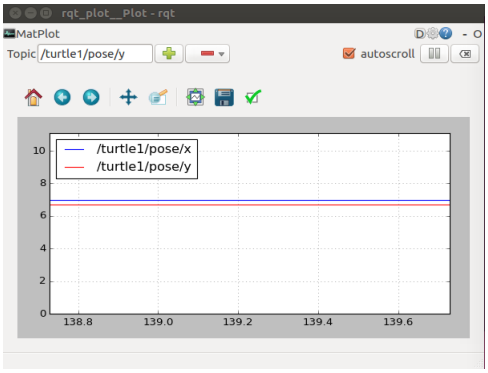

Figure 56 rqt_plot

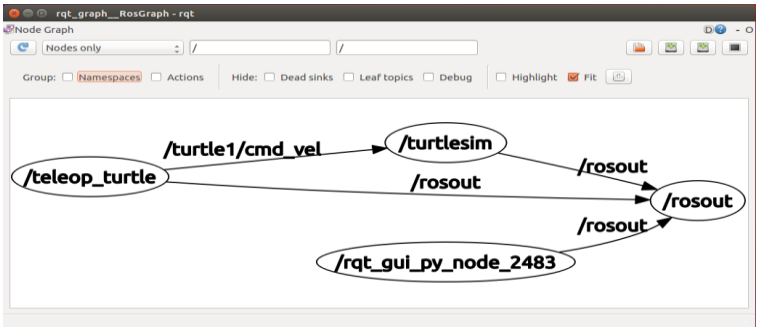

Figure 54 rqt_graph

4.12. ROS Applications

4.12.1. Application 1: Preparation of ROS Environment

The workspace is a folder that contains packages. These packages contain source files. It is useful when various packages are wanted to be compiled (centralized) at the same time.

- **Resource area (src):** Resource area (src folder), packages, projects, etc. Placed. This area also contains the *CMakeLists.txt* file.
- **Compile area (build):** stores cmake and catkin, cache information, configuration, and other buffer files for packages and projects in the build folder.
- **Development area (devel):** It is used to protect compiled programs and test programs without the installation phase.

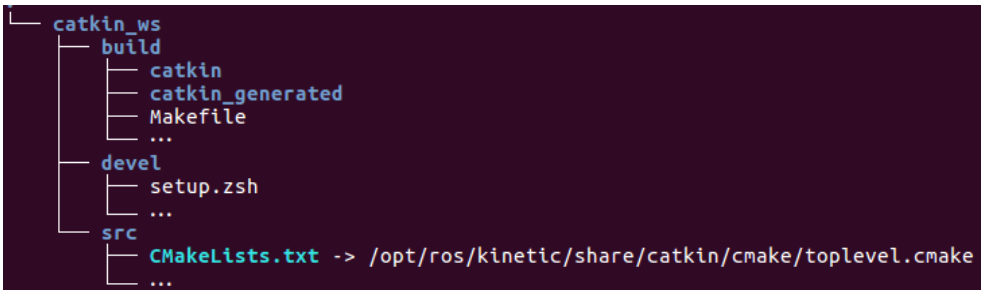

Figure 57 Workspace Folder Structure

Lets Check environment:

```
printenv | grep ROS
```

In order not to make our configuration settings every time:

```
gedit ~/.bashrc
```

The following codes are added to the screen opened in the Gedit editor.

```
<source /opt/ros/kinetic/setup.bash>
<source /home/<kullanıcı_adı>/ros_ws/devel/setup.bash>
```

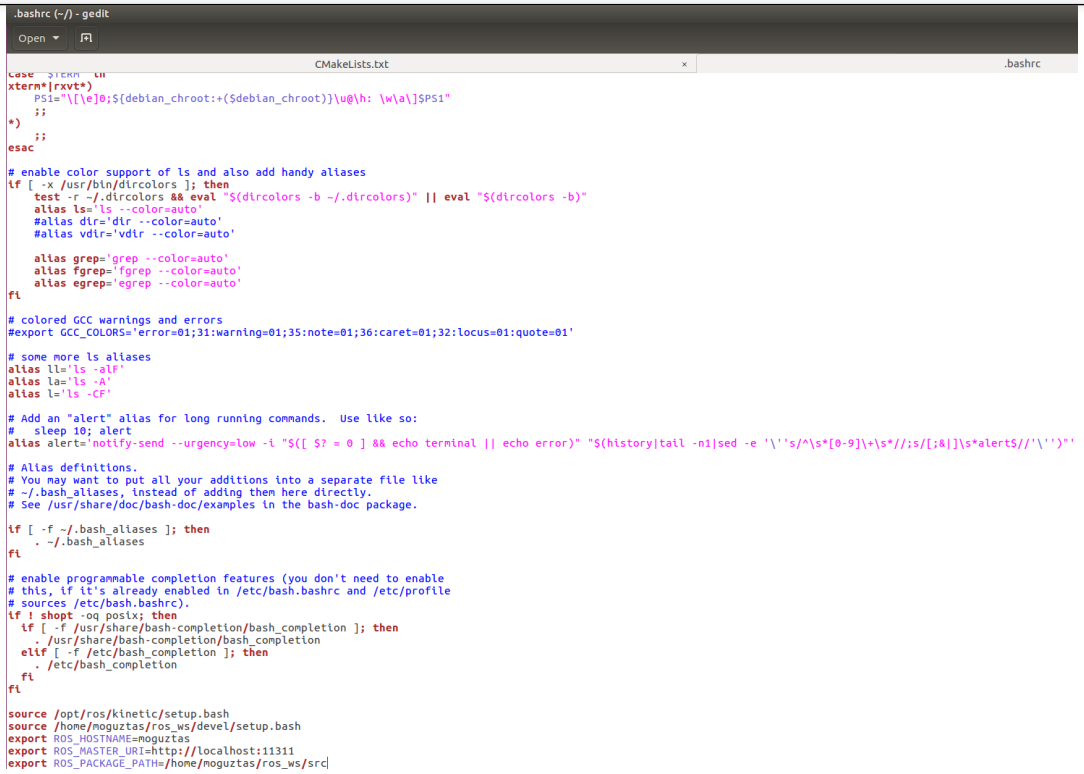

Figure 58 .Bashrc

Note: The terminal must be renewed with the bash command so that changes made in bashrc can be detected in the terminal that was opened previously.

To create work environment:

```
mkdir -p ~/ros_ws/src
cd ~/ros_ws/
catkin_make
```

Note: A block of code that can do the same with catkin_make:

```
cd ~/ros_ws
cd src
catkin_init_workspace
cd ..
```

```
mkdir build
cd build
cmake ../src -DCMAKE_INSTALL_PREFIX=../install -
DCATKIN_DEVEL_PREFIX=../devel
make
```

Note: The following code is written on the terminal screen for the location of the ROS_PACKAGE_PATH configuration variable.

```
echo $ROS_PACKAGE_PATH
/home/(KULLANICI_ADINIZ)/ros_ws/src:/opt/ros/kinetic/s
hare
```

4.12.2. Application 2: Creating a Catkin Package and Getting to Know the ROS Environment

First, go to the src directory in the ros workspace.

```
cd ~/ros_ws/src
```

With the catkin_create_pkg command, a package named beginner_tutorials is linked to std_msgs, roscpp and rospy:

```
catkin_create_pkg beginner_tutorials std_msgs rospy
roscpp
```

Note: This will create a file containing package.xml named beginner_tutorial and a CMakeLists.txt file. The CMakeLists.txt file is partially populated by the catkin_create_pkg command.

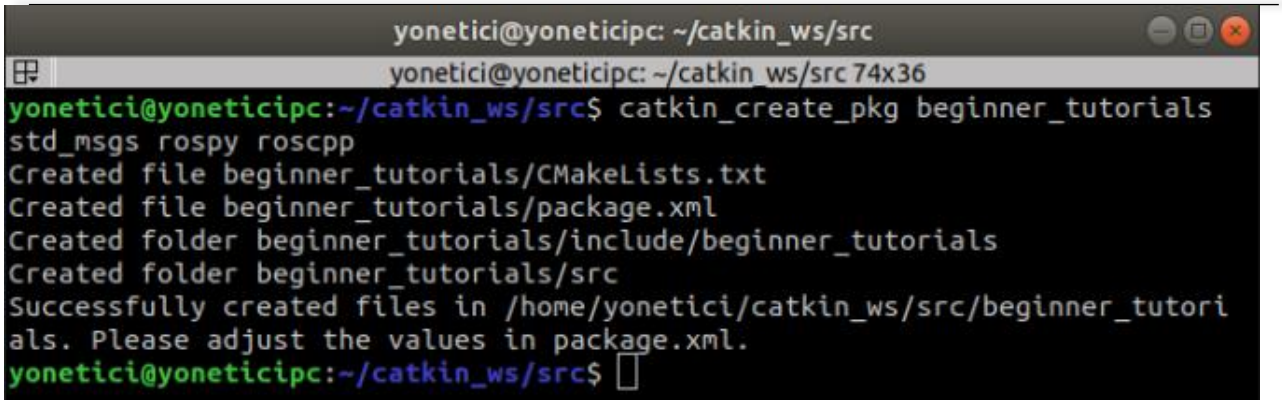

Figure 59 Catkin Package Creation

To see what happens in the beginner tutorials folder:

```
cd beginner_tutorials
ls
```

It will be seen that there will be Cmakelist.txt, include and src folders inside the folder. Let's go to beginner_tutorials src file and write a simple code here:

```
cd src
gedit first_script.cpp
```

Let's make the code we write in the CMakeLists.txt file executable:

```
cd ..
gedit CMakeLists.txt
```

Let's compile workspace.

```
cd ~/ros_ws/
catkin_make
```

Note: When the compilation is completed, build, devel and src subfolders will be installed in the src folder, the package will be ready for use.

first_script.cpp (~/ros_ws/src/beginner_tutorials/src) - gedit

Open Save

```
#include <iostream>

using namespace std;
int main()
{
        cout << "ROS Uygulamali Egitim" << endl;
        return 0;
}
```

Figure 61 first_script.cpp

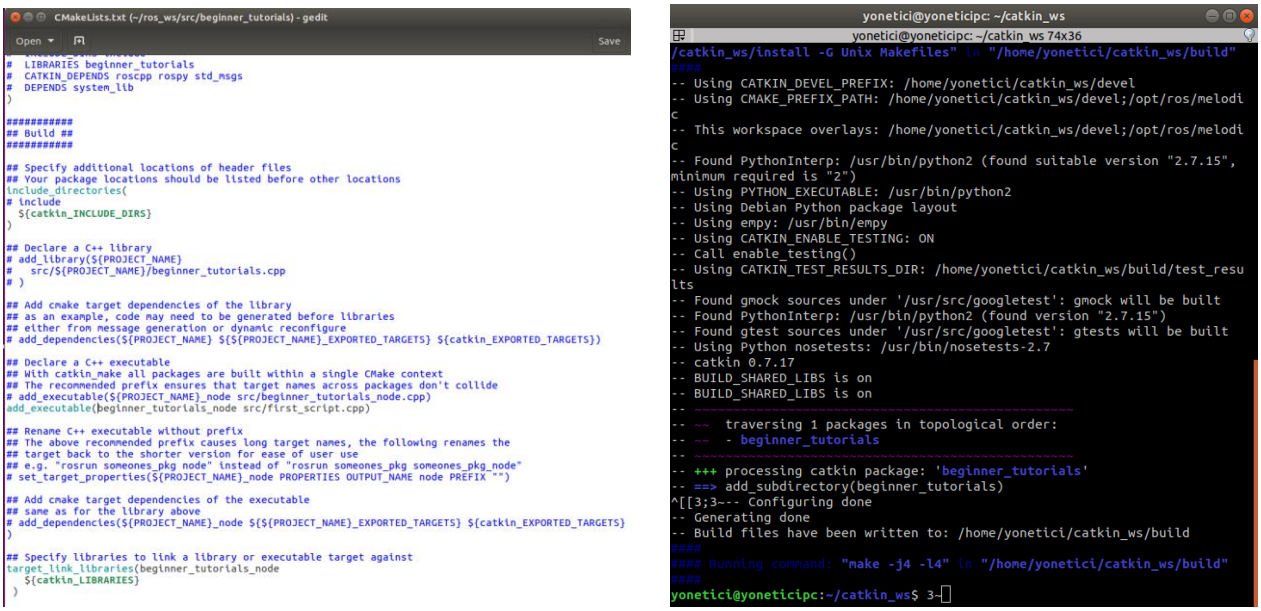

Figure 60 Compiling Catkin.

Two terminals are opened to run the written code. The following code is executed in the first terminal.

```
roscore
```

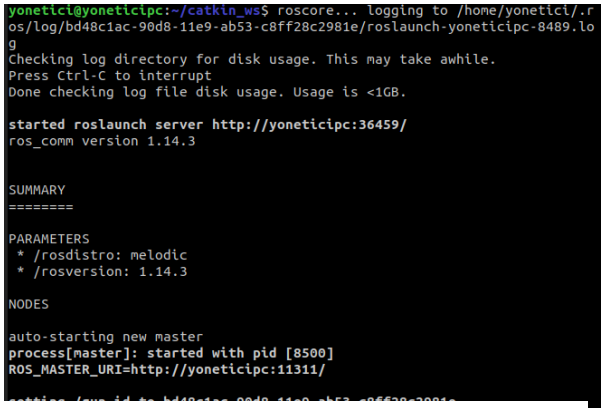

Figure 62 roscore Screen

In the other terminal, the ros package created is run.

```
rosrun beginner_tutorials
beginner_tutorials_node
```

To view the dependencies on the package with the rospack tool:

```
rospack depends1 beginner_tutorials
```

These dependencies are also listed in the package.xml file.

roscd beginner_tutorials
gedit package.xml

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>


<!-- The export tag contains other, unspecified, tags -->
<export>
  <!-- Other tools can request additional information be placed here -->

</export>
</package>
```

Figure 63 Package.xml

Indirect dependencies can be viewed with the rospack tool. For example, to see rospy dependencies:

```
rospack depends1 rospy
```

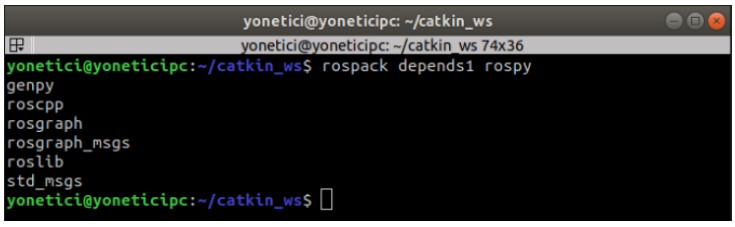

Figure 64 rospack Example

To see all the dependencies in the package:

```
rospack depends beginner_tutorials
```

```
yonetici@yoneticipc: ~/catkin_ws
yonetici@yoneticipc: ~/catkin_ws 74x36
yonetici@yoneticipc:~/catkin_ws$ rospack depends beginner_tutorials
cpp_common
rostime
roscpp_traits
roscpp_serialization
catkin
genmsg
genpy
message_runtime
gencpp
geneus
gennodejs
genlisp
message_generation
rosbuild
rosconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
ros_environment
rospack
roslib
rospy
yonetici@yoneticipc:~/catkin_ws$
```

Figure 65 All rospack Dependencies

4.12.3. Application 3: TurtleSim

For TurtleSim application:

```
sudo apt-get install ros-kinetic-ros-tutorials
```

To start the ROS Master:

```
roscore
```

To run TurtleSim:

```
rosrun turtlesim
turtlesim_node
```

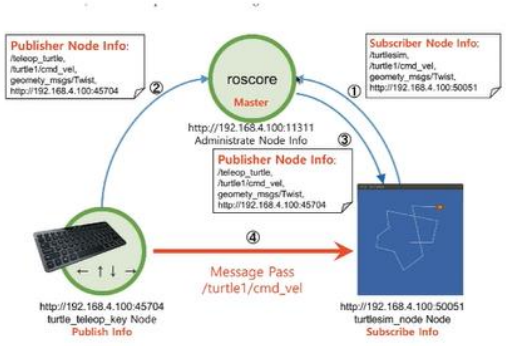

Figure 66 Turtlebot Teleoperation

For keyboard control:

```
rosrun turtlesim turtle_teleop_key
```

Write the codes below in order. Sample output is provided for each code.

```
rosservice list
```

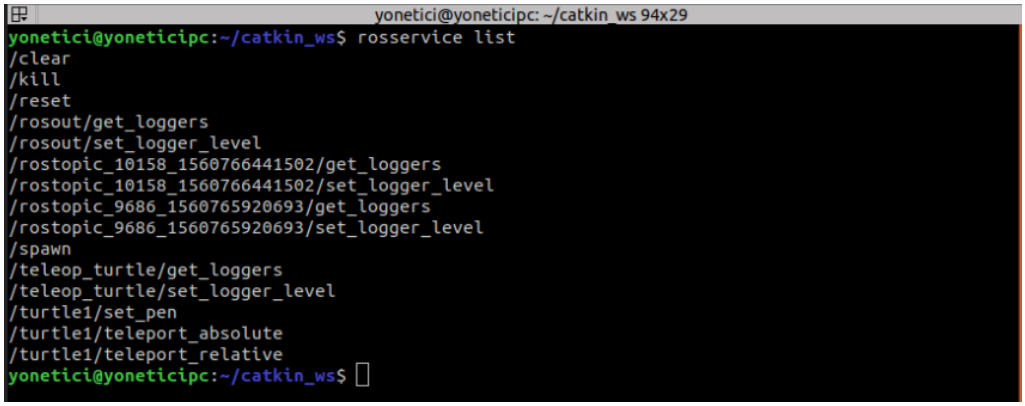

Figure 67 rosservice List

```
rosservice type/spawn
```

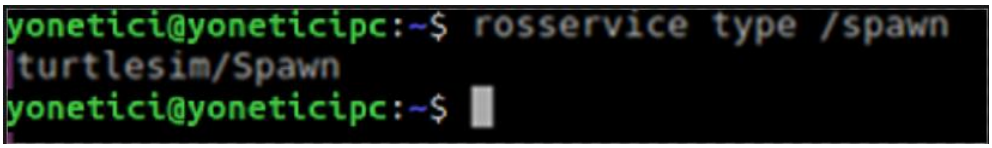

Figure 68 TurtleSim creation

```
rossrv show turtlesim/Spawn
```

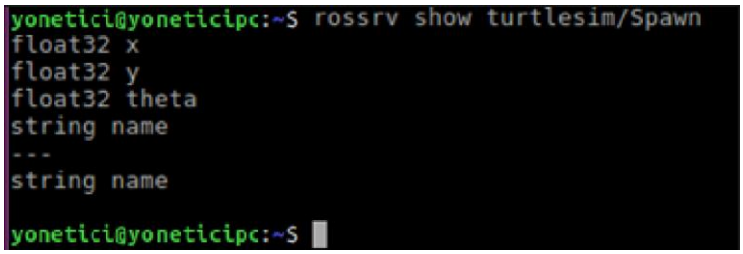

Figure 69 TurtleSim/Spawn Service Monitoring

```
rosservice call /spawn 3 3 0 new_turtle
```

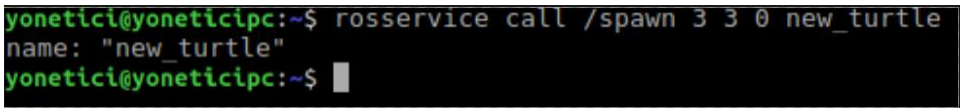

Figure 70 Calling New Service

```
rosparam list
```

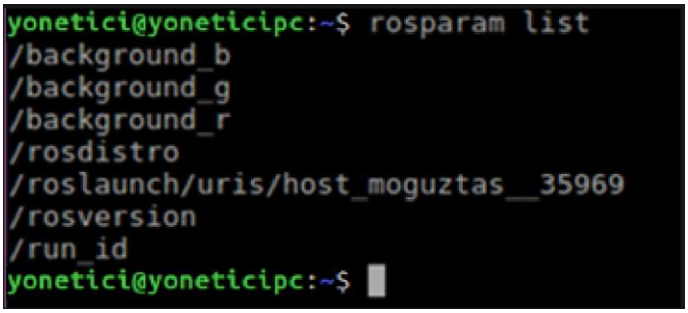

Figure 71 Ros Parameter List

```
rosparam get /background_b
```

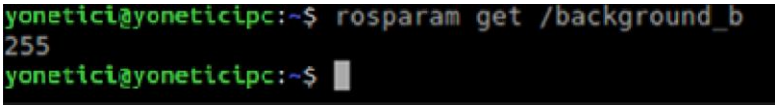

Figure 72 Backtround Modification

```
rosparam set /background_b 10
rosparam get /background_b 10
```

Figure 73 TurtleSim Background Modification Example

```
rosservice call /clear
```

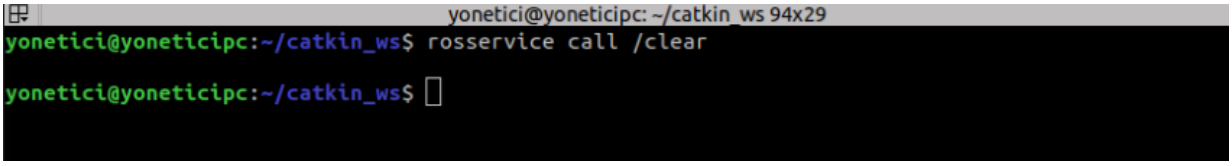

Figure 74 Closing Services

4.12.4. Application 4: Creating Messages and Services

Creating Messages:

Let's create the msg folder in the beginner_tutorials folder in the workspace.

```
roscd beginner_tutorials
mkdir msg
```

Let's create our message file and show the file we created in the terminal.

```
echo "int64 num" > msg/Num.msg
rosmsg show beginner_tutorials/Num
```

Let's edit our package.xml file.

```
roscd beginner_tutorials
gedit package.xml
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

Let's edit our CMakeLists.txt file as follows.

```
gedit CMakeLists.txt
find_package(catkin REQUIRED COMPONENTS
...
message_generation
)

catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...
)
```

```
add_message_files(
FILES
Num.msg
)
generate_messages(
DEPENDENCIES
std_msgs
)
```

Lets compile workspace.

```
cd ~/ros_ws
catkin_make
```

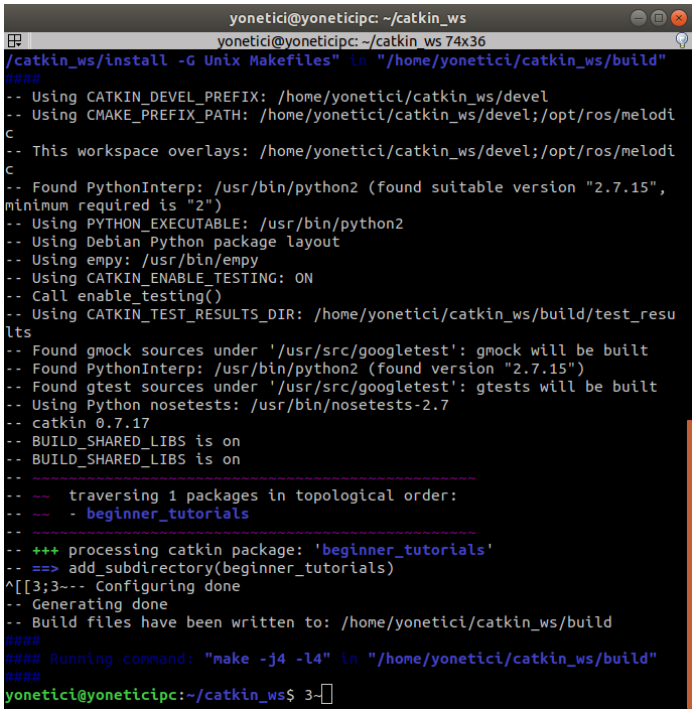

Figure 75 Compiling atkin

Creating Services:

Let's create the srv folder in the beginner_tutorials folder in the workspace.

```
roscd beginner_tutorials
mkdir srv
```

Let's create our srv file and show the file we created in the terminal.

```
roscp rospy_tutorials AddTwoInts.srv
srv/AddTwoInts.srv
rossrv show beginner_tutorials/AddTwoInts
```

Not: srv files are like msg files, except they contain two partitions.

Let's edit our package.xml file.

```
roscd beginner_tutorials
gedit package.xml
    <build_depend>message_generation</build_depend>
    <exec_depend>message_runtime</exec_depend>
```

Let's edit our CMakeLists.txt file as follows.

```
gedit CMakeLists.txt
find_package(catkin REQUIRED COMPONENTS
...
message_generation
)

catkin_package(
...
```

```
CATKIN_DEPENDS message_runtime ...
...
)

add_service_files(
FILES
AddTwoInts.srv
)
generate_messages(
DEPENDENCIES
std_msgs
)
```

Compile workspace.

```
cd ~/ros_ws
catkin_make
```

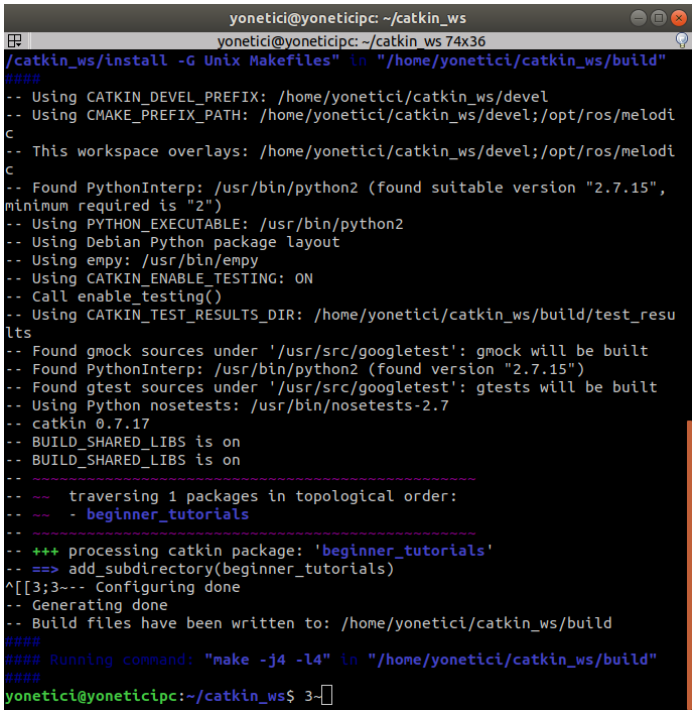

Figure 76 Compiling Caktin

4.12.5. Application 5: Publisher-Subscriber Application

For C++:

Let's go to the src folder under beginner_tutorials.

```
roscd beginner_tutorials/src
```

Let's create our Publisher file.

```
gedit talker.cpp
```

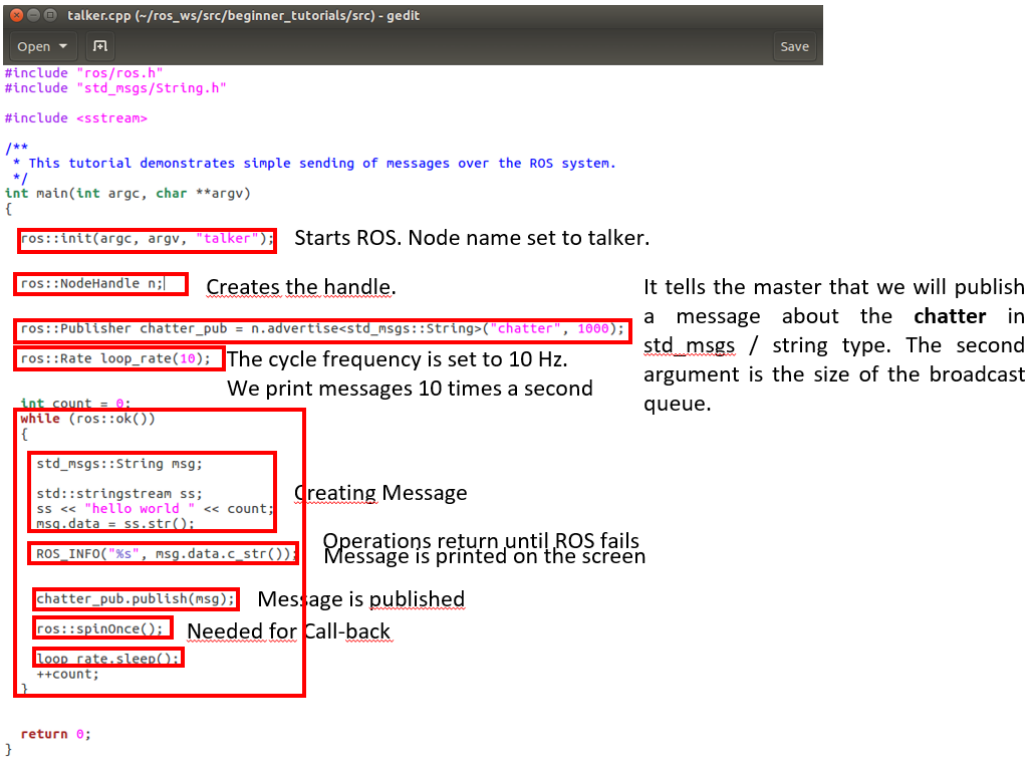

Figure 77 Talker.cpp and Details

Let's create our subscriber file.

```
gedit listener.cpp
```

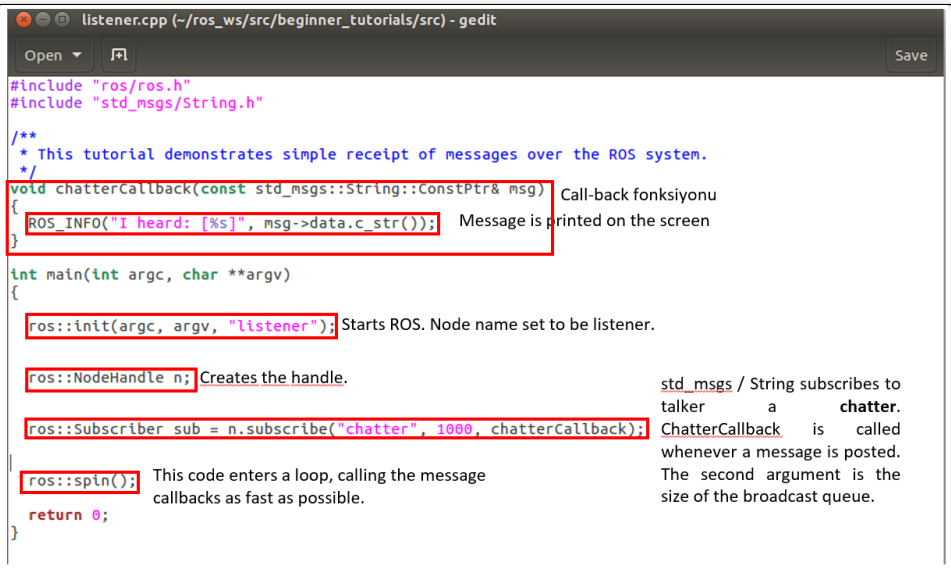

Figure 78 Listener.cpp and Details

Let's edit our CMakeLists.txt file..

```
roscd beginner_tutorials
gedit CMakeLists.txt
```

Compile workspace.

```
cd ~/ros_ws
catkin_make
```

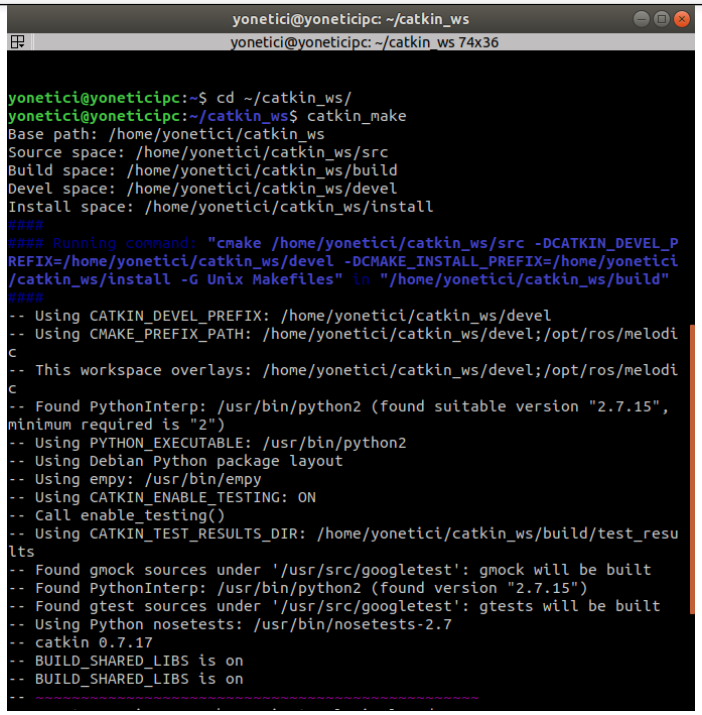

Figure 79 Compiling Catkin

For Python:

Let's go to beginner_tutorials folder and create scripts folder.

```
roscd beginner_tutorials/src
mkdir scripts
cd scripts
```

Let's create Publisher file.

```
gedit talker.py
```

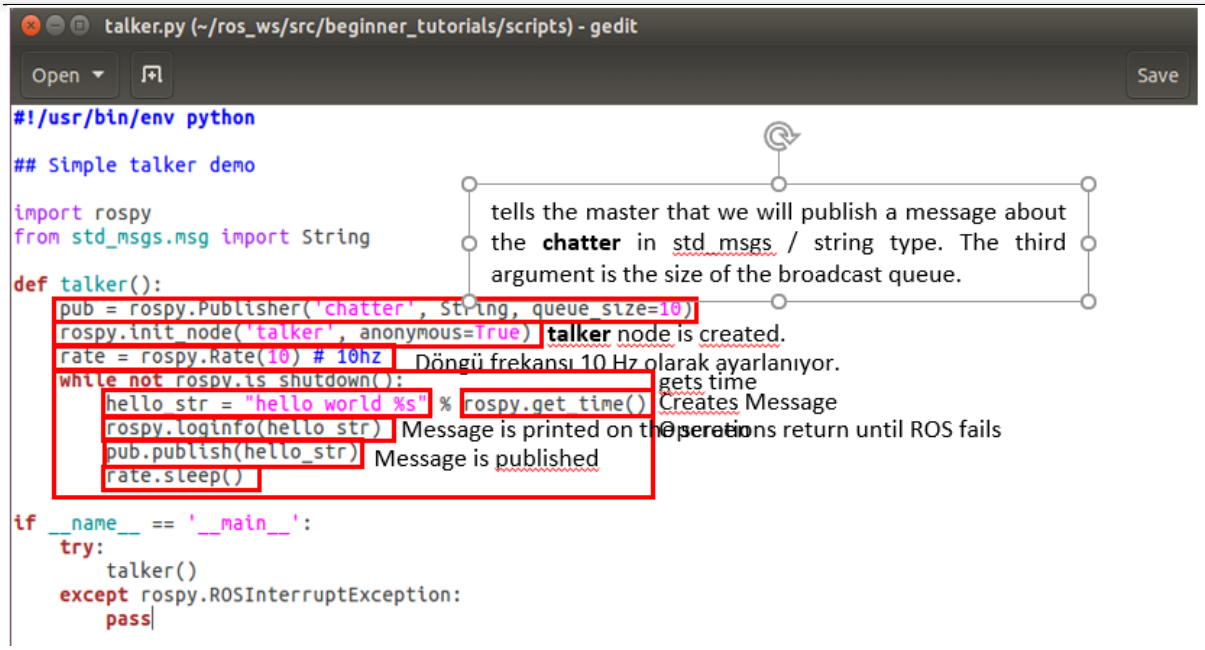

Figure 80 talker.py and Details

Let's create subscriber file.

```
gedit listener.py
```

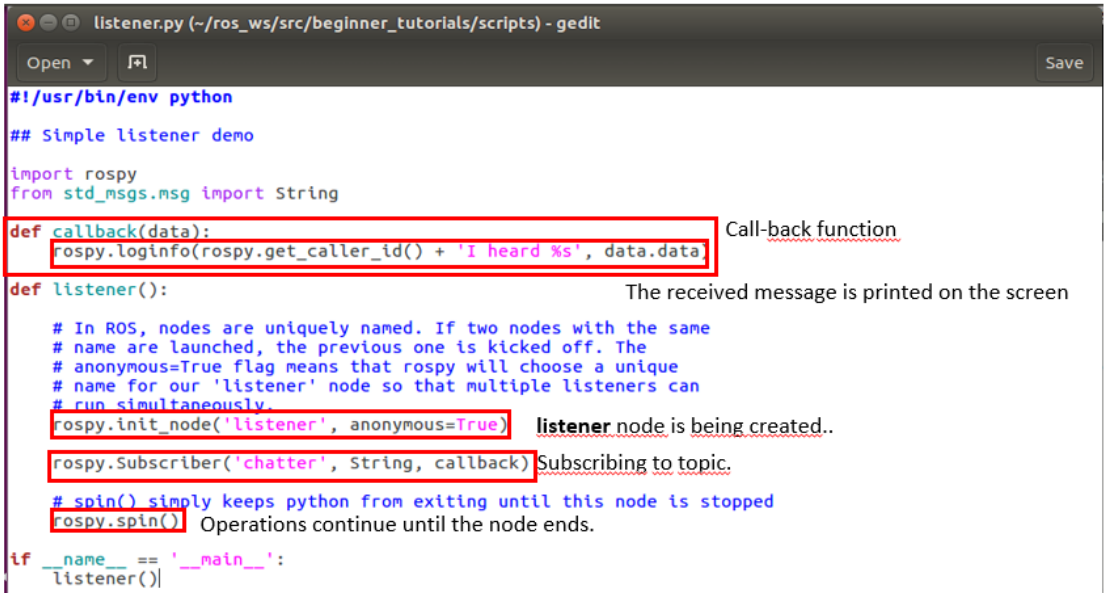

Figure 81 Listener.py and Details

Let's make executable files.

```
ls
chmod +x listener.py
```

Compile workspace.

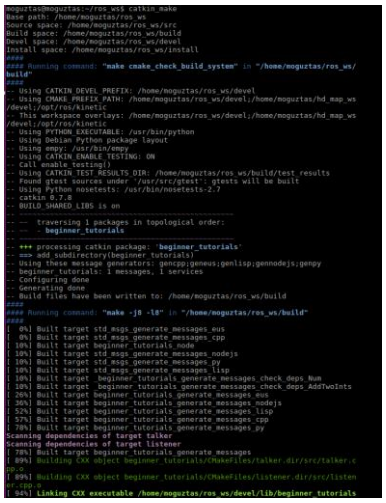

Figure 82 Compiling Catkin

```
cd ~/ros_ws
catkin_make
```

Run the application:

- Terminal 1:
  - Roscore
- Terminal 2:
  - (C++) rosrun beginner_tutorials talker
  - (Python) rosrun beginner_tutorials talker.py
- Terminal 3:
  - (C++) rosrun beginner_tutorials listener
  - (Python) rosrun beginner_tutorials listener.py

```
moguztas@moguztas:~/ros_ws$ rosrun beginner_tutorials talker
[ INFO] [1567209367.884622634]: hello world 0
[ INFO] [1567209367.984790365]: hello world 1
[ INFO] [1567209368.084775385]: hello world 2
[ INFO] [1567209368.184757122]: hello world 3
[ INFO] [1567209368.284752723]: hello world 4
[ INFO] [1567209368.384781284]: hello world 5
[ INFO] [1567209368.484788813]: hello world 6
[ INFO] [1567209368.584788005]: hello world 7
[ INFO] [1567209368.684791028]: hello world 8
[ INFO] [1567209368.784744496]: hello world 9
[ INFO] [1567209368.884766553]: hello world 10
[ INFO] [1567209368.984751599]: hello world 11
[ INFO] [1567209369.084771260]: hello world 12
[ INFO] [1567209369.184755739]: hello world 13
[ INFO] [1567209369.284732317]: hello world 14
[ INFO] [1567209369.384752488]: hello world 15
[ INFO] [1567209369.484772846]: hello world 16
[ INFO] [1567209369.584798485]: hello world 17
[ INFO] [1567209369.684763541]: hello world 18
[ INFO] [1567209369.784781674]: hello world 19
[ INFO] [1567209369.884801013]: hello world 20
[ INFO] [1567209369.984798947]: hello world 21
[ INFO] [1567209370.084707976]: hello world 22
[ INFO] [1567209370.184756887]: hello world 23
[ INFO] [1567209370.284791587]: hello world 24
[ INFO] [1567209370.384794507]: hello world 25
```

Figure 83 Terminal 2

```
moguztas@moguztas:~/ros_ws$ rosrun beginner_tutorials listener.py
[INFO] [1567209575.599204]: /listener_16717_1567209575367I heard hello world 165
[INFO] [1567209575.699208]: /listener_16717_1567209575367I heard hello world 166
[INFO] [1567209575.799234]: /listener_16717_1567209575367I heard hello world 167
[INFO] [1567209575.899221]: /listener_16717_1567209575367I heard hello world 168
[INFO] [1567209575.999226]: /listener_16717_1567209575367I heard hello world 169
[INFO] [1567209576.099204]: /listener_16717_1567209575367I heard hello world 170
[INFO] [1567209576.199212]: /listener_16717_1567209575367I heard hello world 171
[INFO] [1567209576.299228]: /listener_16717_1567209575367I heard hello world 172
[INFO] [1567209576.399248]: /listener_16717_1567209575367I heard hello world 173
[INFO] [1567209576.499223]: /listener_16717_1567209575367I heard hello world 174
[INFO] [1567209576.599137]: /listener_16717_1567209575367I heard hello world 175
[INFO] [1567209576.699225]: /listener_16717_1567209575367I heard hello world 176
[INFO] [1567209576.799226]: /listener_16717_1567209575367I heard hello world 177
[INFO] [1567209576.899234]: /listener_16717_1567209575367I heard hello world 178
[INFO] [1567209576.999163]: /listener_16717_1567209575367I heard hello world 179
[INFO] [1567209577.099210]: /listener_16717_1567209575367I heard hello world 180
[INFO] [1567209577.199122]: /listener_16717_1567209575367I heard hello world 181
[INFO] [1567209577.299158]: /listener_16717_1567209575367I heard hello world 182
[INFO] [1567209577.399200]: /listener_16717_1567209575367I heard hello world 183
[INFO] [1567209577.499195]: /listener_16717_1567209575367I heard hello world 184
[INFO] [1567209577.599238]: /listener_16717_1567209575367I heard hello world 185
[INFO] [1567209577.699100]: /listener_16717_1567209575367I heard hello world 186
[INFO] [1567209577.799165]: /listener_16717_1567209575367I heard hello world 187
[INFO] [1567209577.899168]: /listener_16717_1567209575367I heard hello world 188
[INFO] [1567209577.999260]: /listener_16717_1567209575367I heard hello world 189
[INFO] [1567209578.099261]: /listener_16717_1567209575367I heard hello world 190
```

Figure 84 Terminal 3

4.12.6. Application 6: Service-Client Application

For C++:

Let's go to the src folder under beginner_tutorials.

```
roscd beginner_tutorials/src
```

Let's create Server file.

```
gedit add_two_ints_server.cpp
```

add_two_ints_server.cpp (~/ros_ws/src/beginner_tutorials/src) - gedit

Open Save

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.sum = req.a + req.b;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
  ROS_INFO("sending back response: [%ld]", (long int)res.sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

Figure 85 add_two_ints_server.cpp

Let’s create Client file.

```
gedit add_two_ints_client.cpp
```

add_two_ints_client.cpp (~/ros_ws/src/beginner_tutorials/src) - gedit

Open Save

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

Figure 86 add_two_ints_client.py

Let's edit CMakeLists.txt file.

```
roscd beginner_tutorials
gedit CMakeLists.txt
```

Compile workspace.

```
cd ~/ros_ws
```

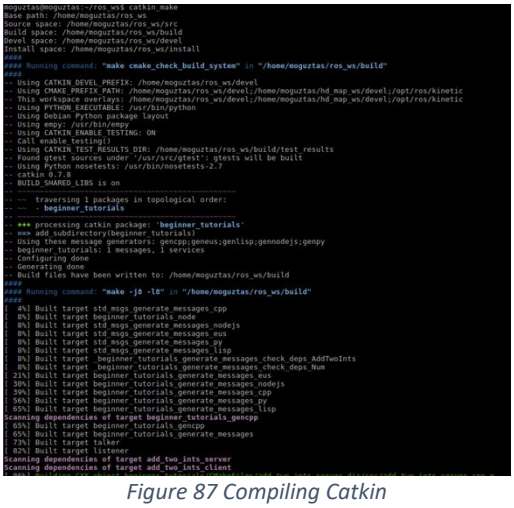

Figure 87 Compiling Catkin

```
catkin_make
```

For Python:

Let's go to beginner_tutorials folder and create scripts folder.

```
roscd beginner_tutorials/src
```

Let's create Server file.

```
gedit add_two_ints_server.cpp
```

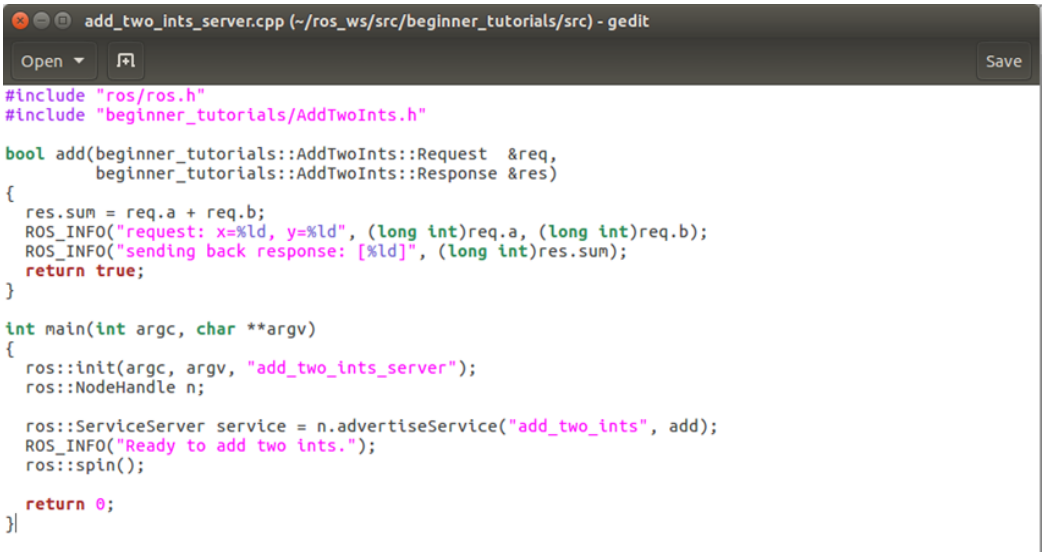

Figure 88 add_two_ints_server.py

Let's create Client file.

```
gedit add_two_ints_client.cpp
```

add_two_ints_client.cpp (~/ros_ws/src/beginner_tutorials/src) - gedit

Open Save

```
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client");
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
  beginner_tutorials::AddTwoInts srv;
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);
  if (client.call(srv))
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
  }

  return 0;
}
```

Figure 89 add_two_ints_client.py

Let's make our files executable.

```
chmod +x add_two_ints_server.py
chmod +x add_two_ints_client.py
ls
```

Compile workspace.

```
cd ~/ros_ws
catkin_make
```

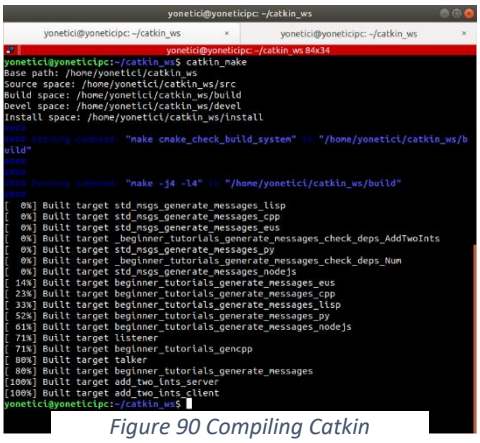

Figure 90 Compiling Catkin

Running Application:

- Terminal 1:
  - Roscore
- Terminal 2:
  - (C++) rosrun beginner_tutorials add_two_ints_server
  - (Python) rosrun beginner_tutorials add_two_ints_server.py
- Terminal 3:
  - (C++) rosrun beginner_tutorials add_two_ints_server
  - (Python) rosrun beginner_tutorials add_two_ints_server.py

Figure 93 Terminal 1

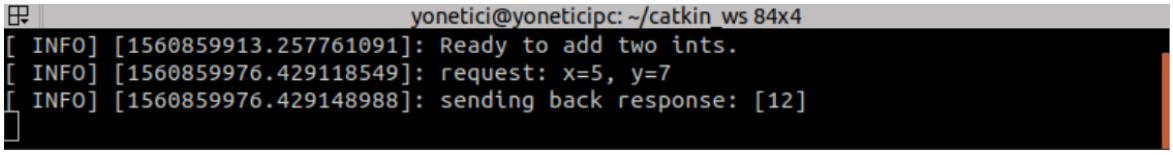

Figure 91 Terminal 2

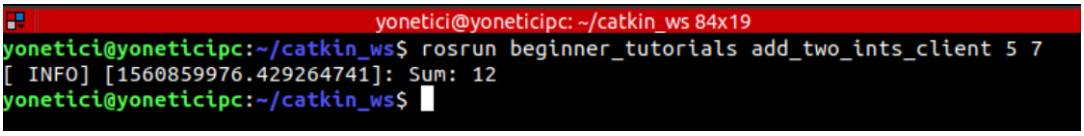

Figure 92 Terminal 3

4.12.7. Application 7: Saving and Playing Data

Let's run roscore

```
roscore
```

Let's open TurtleSim.

```
rosrun turtlesim turtlesim_node
```

Let's open the keyboard control node.

```
rosrun turtlesim turtle_teleop_key
```

Open the folder named bagfiles under the beginner_tutorials folder.

```
mkdir ~/bagfiles
cd ~/bagfiles
```

To save all published topics:

```
roscd beginner_tutorials/src
```

To record some topics:

```
gedit add_two_ints_server.cpp
```

Let's move our robot with the help of the keyboard as given in figures.

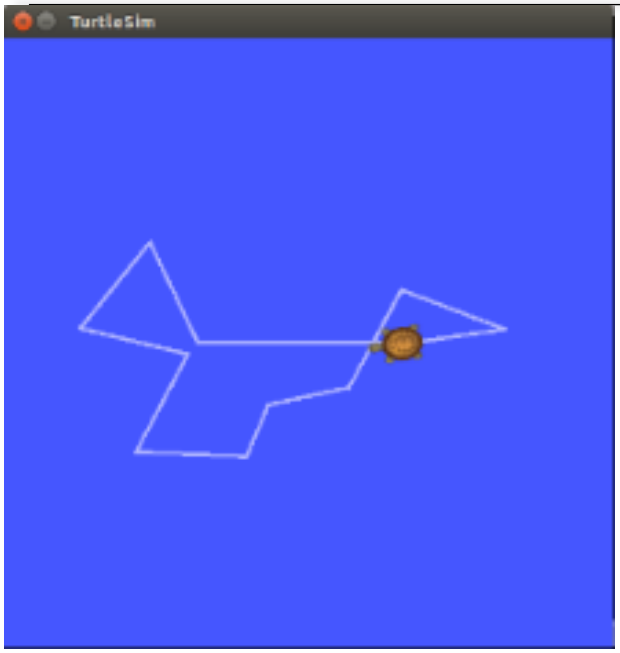

Figure 94 TurtleSim Teleoperation

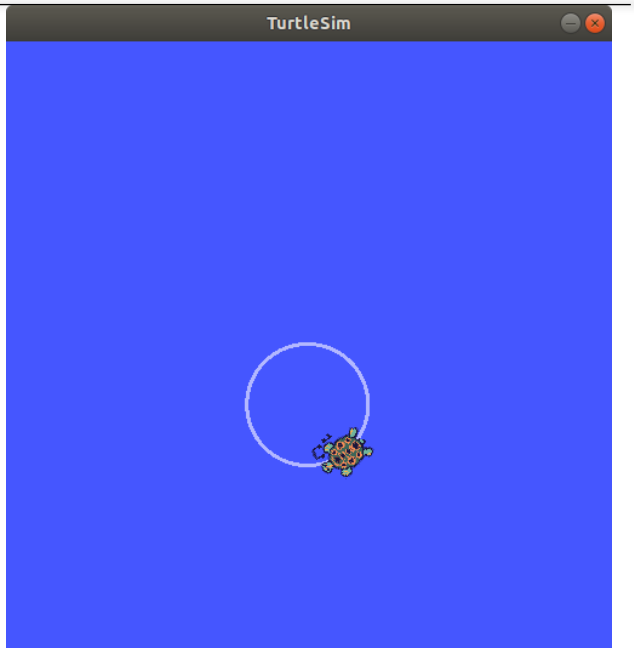

Figure 95 TurtleSim on constant velocity

Let's check the content of our bag file.

```
rosbag info bag_file
```

```
yonetici@yoneticipc:~/catkin_ws$ cd
yonetici@yoneticipc:~/bagfiles$ rosbag record -a

[ INFO] [1560861409.542054607]: Recording to 201
9-06-18-15-36-49.bag.
[ INFO] [1560861409.543464238]: Subscribing to /
turtle1/color_sensor
[ INFO] [1560861409.545428123]: Subscribing to /
turtle1/cmd_vel
[ INFO] [1560861409.547594081]: Subscribing to /
rosout
[ INFO] [1560861409.550016397]: Subscribing to /
rosout_agg
[ INFO] [1560861409.552390785]: Subscribing to /
turtle1/pose
```

Figure 96 rosbag record

```
yonetici@yoneticipc:~/bagfiles$ rosbag info 2019
-06-18-15-36-49.bag
path:        2019-06-18-15-36-49.bag
version:     2.0
duration:    53.4s
start:       Jun 18 2019 15:36:49.55 (1560861409
.55)
end:         Jun 18 2019 15:37:42.97 (1560861462
.97)
size:        463.0 KB
messages:    6654
compression: none [1/1 chunks]
types:       rosgraph_msgs/Log [acffd30cd6b6de30
f120938c17c593fb]
             turtlesim/Color   [353891e354491c51
aabe32df673fb446]
             turtlesim/Pose    [863b248d5016ca62
ea2e895ae5265cf9]
topics:      /rosout                      4 msgs
  : rosgraph_msgs/Log (2 connections)
             /turtle1/color_sensor   3325 msgs
  : turtlesim/Color
             /turtle1/pose           3325 msgs
  : turtlesim/Pose
```

Figure 97 rosbag content

Let's play the Bag File

```
yonetici@yoneticipc:~/ros_ws/src/beginner_tutorials/bagfiles$ rosbag play 2019-08-31-03-21-38.bag
[ INFO] [1567211137.277837787]: Opening 2019-08-31-03-21-38.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
[RUNNING]  Bag Time: 1567210939.989698   Duration: 41.603941 / 76.436171
```

-O argument tells the rosbag record command to just follow and write these two

Figure 98 execution of rosbag

topics in a file called subset.bag.

5. Gazebo

5.1. What is Gazebo?

Robust physics engine for simulating robots in indoor and outdoor environments, provides the necessary infrastructure with high quality graphics and graphic interface.

Figure 99 Gazebo Simulation Platform

Gazebo:

- has high performance physic engines like ODE, Bullet, Simbody ve DART
- creates a realistic environment with OGRE
- enables sensor ve sensor data usage,
- usage of existing robot models or the possibility to create special robot models

5.2. Gazebo Components

A simulation performed with Gazebo includes following items:

- World Files
- Model Files
- Environment Variables
- Gazebo server
- Gazebo client

- Plugins

5.2.1. Word Files

World Files, define environment models in simulation environment like;

- robot,
- sensor,
- light,
- object

These files are created using the definition format named SDF and generally have the .world extension.

These files are created using the definition format named SDF and generally have a .world extension.

Various world files come with the Gazebo installation and

- <install_path>/share/gazebo-<version>/worlds

Include these.

It is also possible to create user-defined world files using the SDF definition format.

“Empty.world”, which is a sample world file, has the following content..

```
<?xml version="1.0" ?>
<sdf version="1.5">
    <world name="default">
    <!-- A global light source -->
```

```
            <include>
                <uri>model://sun</uri>
            </include>
        <!-- A ground plane -->
            <include>
                <uri>model://ground_plane</uri>
            </include>
        </world>
    </sdf>
```

5.2.2. Model Files

model files use SDF definition format like World files and only contain

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

Style tags.

The purpose of using these files is to make models reusable and to simplify world files.

A model created can be used in the world file as follows.

```
    <include>
            <uri>model://model_file_name</uri>
    </include>
```

For the simulation environment, model files that are included in the online database or included with the installation in old Gazebo versions can be used, or user-defined model files can be used.

5.2.3. Environment Variables

Gazebo uses various environment variables to locate and access files and establish the relationship between server and client. The environment variables come compiled with the installation.

The default environment variables are located in the shell script (.sh file) below.

- <install_path>/share/gazebo/setup.sh

To change environmental variables that Gazebo will consider flollowing code

- source <install_path>/share/gazebo/setup.sh

should be referenced and should be modified optionally.

An example script file content is shown below.

```
    export GAZEBO_MASTER_URI=http://localhost:11345
    export
GAZEBO_MODEL_DATABASE_URI=http://gazebosim.org/models
    export GAZEBO_RESOURCE_PATH=/usr/share/gazebo-
7:/usr/share/gazebo_models:${   GAZEBO_RESOURCE_PATH}
    export GAZEBO_PLUGIN_PATH=/usr/lib/x86_64-linux-
gnu/gazebo-7/plugins:$ {GAZEBO_PLUGIN_PATH}
    export GAZEBO_MODEL_PATH=/usr/share/gazebo-
7/models:${   GAZEBO_MODEL_PATH}
```

```
export
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/lib/x86_64-
linux-gnu/gazebo- 7/plugins
```

5.2.4. Gazebo Server

The Gazebo server is the part of Gazebo that takes over the entire processing load. It parses a world file given to it and then simulates the world using a physics and sensor engine.

Gazebo server does not contain any graphic interface.

Gazebo server can be started simply with command

- gzserver

and also

- gzserver worlds/empty.world

command can also start a world file as well(in this example, a world file that comes with Gazebo).

5.2.5. Gazebo Client

The Gazebo client takes over the task of connecting to a working Gazebo server and visualizing the elements of the environment.

It also offers the user the opportunity to make changes to a working simulation.

Gazebo server can simply be run with

- gzclient

command in terminal.

5.2.6. Plugins

The plugins are compiled as common library files and are treated as necessary code snippets. The plugins are available in all Gazebo over standard C ++ classes.

The plugins allow to control the direction of Gazebo and can be added to and removed from a working system.

Plugins can be used when something is desired to be programmatically modified in the simulation (moving models, adding new models when certain conditions are met, etc.), when a fast interface is requested for Gazebo, and the created plug-in is desired to be useful for others.

Currently there are six types of plugins exist.

- World
- Model
- Sensor
- System
- Visual
- Interface

Each plugin type is managed by a different Gazebo component. For example, a model plugin is added to a specific model in Gazebo and controls that model.

The plug-in type should be selected according to the desired functionality. For example;

- A pyhsic plugin for controlling world properties engine environment limunation etc. dünya özelliklerini kontrol etmek için bir dünya eklentisi,

- A model plugin to check the condition of joints and model,
- A sensor plugin to get sensor information and check sensor properties

Plugins can be loaded from the terminal command line as follows;

- gzserver -s <plugin_filename>

Also, it can be loaded in the SDF file by specifying as follows;

```
<plugin name="gazebo_ros_control"
filename="libgazebo_ros_control.so">
        <robotNamespace>/MYROBOT</robotNamespace>
</plugin>
```

installed.

5.3. Gazebo Applications

5.3.1. Gazebo Installation

Gazebo is an independent project used by ROS. Usually, the newest Gazebo version available at the beginning of each ROS release cycle is officially selected to be fully integrated and supported.

Packages can be used to install Gazebo on Ubuntu. There are two main sources (packages.ros.org and packages.osrfoundation.org) that host Gazebo packages.

- On packages.ros.org
  - ROS Indigo: Gazebo 2.x
  - ROS Kinetic: Gazebo 7.x
  - ROS Lunar: Gazebo 7.x
  - ROS Melodic: Gazebo 9.x
- On packages.osrfoundation.org

- gazebo 7.x (gazebo7 package name)
- gazebo 8.x (gazebo8 package name)
- gazebo 9.x (gazebo9 package name)

packages are available. any of these sources can be used to install Gazebo.

For users who need to run a specific version of ROS and want to use all the packages related to Gazebo ROS ready to use, it is recommended to use the Gazebo version available at packages.ros.org. For example, ROS Kinetic hosts and uses the 7.x version of Gazebo.

If needed, it is possible to use a certain Gazebo and ROS version together, apart from the recommended options.However, in this case, any ROS Ubuntu package related to Gazebo is not available from the ROS distribution source.

Equivalent of packages can be loaded from OSRF source, but all other software must be created from the source using catkin_workspaces.

Also, if there is a Gazebo version that comes with the current ROS installation, it may be necessary to uninstall this version first when a different Gazebo version is desired.

5.3.2. Running Gazebo

From Terminal

- gazebo

command can start Gazebo. As can be seen in the graphic interface, it is possible to modify the running simulation to a certain extent thanks to the Gazebo client.

5.3.3. Preparing Workspace

In order to perform ROS-Gazebo studies, Catkin working environment can be prepared.

On the “catkin_workspace” we created earlier:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ~/catkin_ws
catkin_make
```

Let's create our package for the sample work to be carried out with the commands.

5.3.4. Creating World

While creating the world to be used in the simulation environment, the ready world files in the Gazebo database can be used as well as the user can create their own world.

World File can be created;

- via graphical interface,
- manually with SDF format compatible descriptions

Let's start Gazebo by running the following commands from the terminal in order to create the world through the graphic interface.

```
cd
gazebo
```

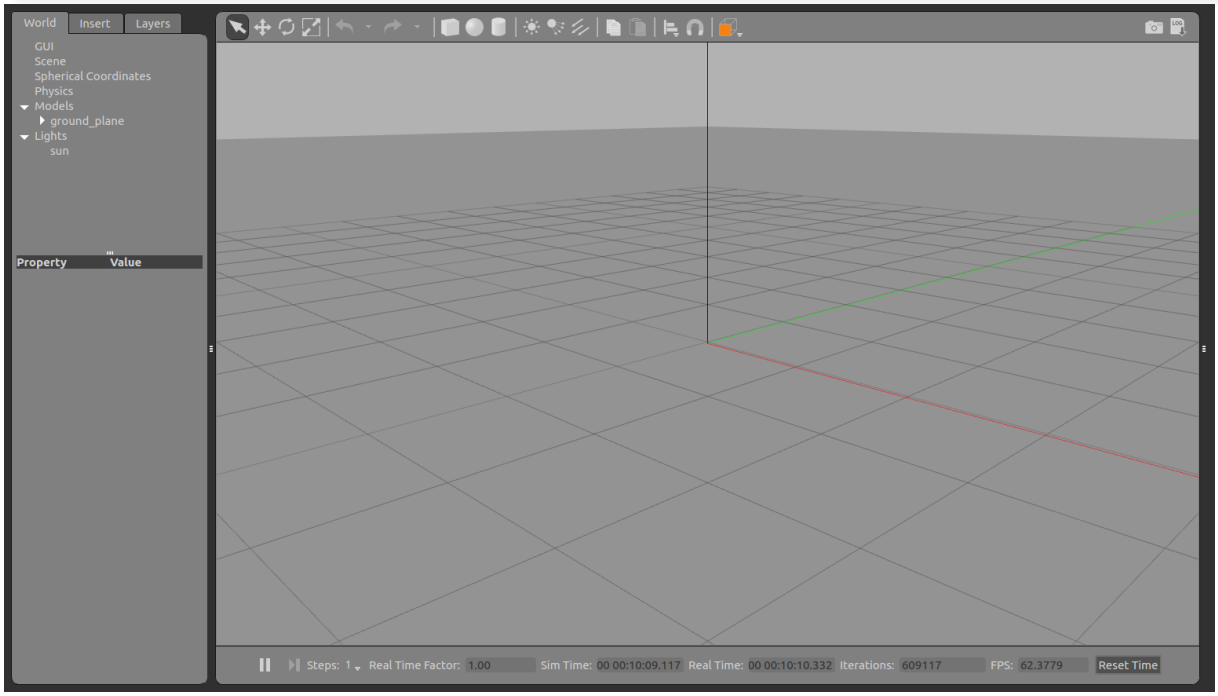

Figure 100 Gazebo workspace

As can be seen in the World section on the left, there is "ground_plane" as the model and "sun" as the Lights. To add a new model, simple objects can be added from the "Simple Shapes" section as seen below, or the model can be added from the model database in the "Insert" section.

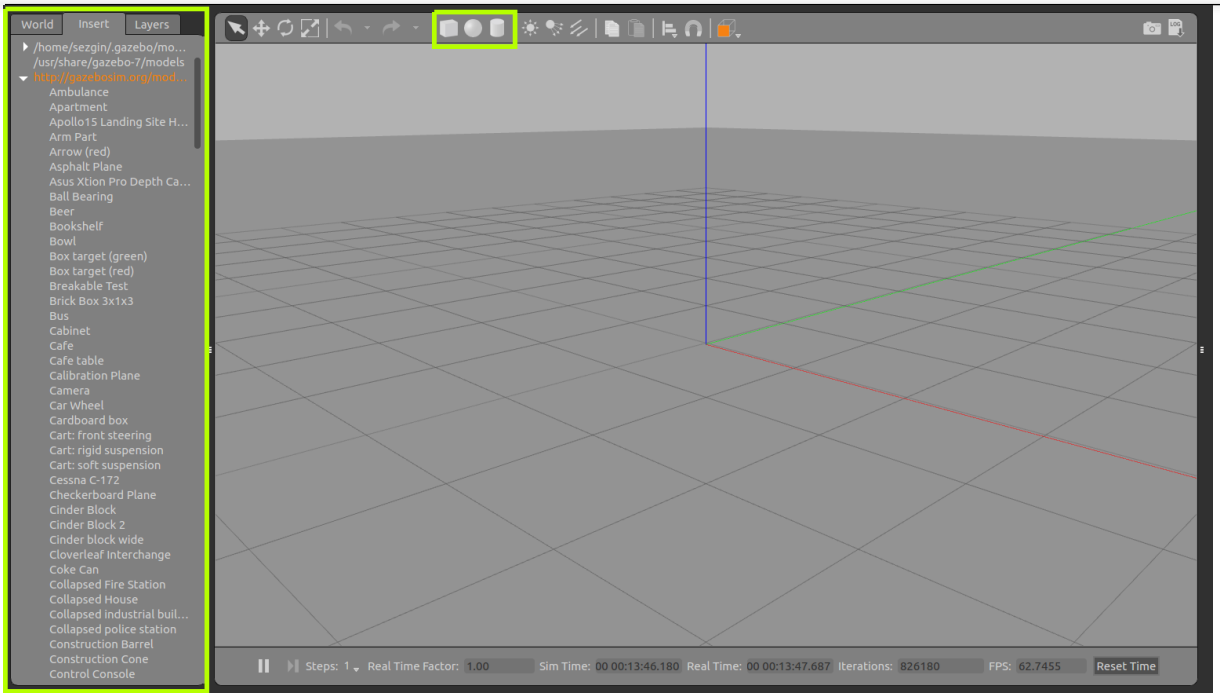

Figure 101 Adding Models

When you want to make changes on the added models, you can change the position of the model with the “Translation” option in the upper toolbar, the position of the model with the “Rotation” section, and the size of the model with the “Scale” option. Likewise, while the relevant model is selected, the position and position of the model can be changed from the options on the left.

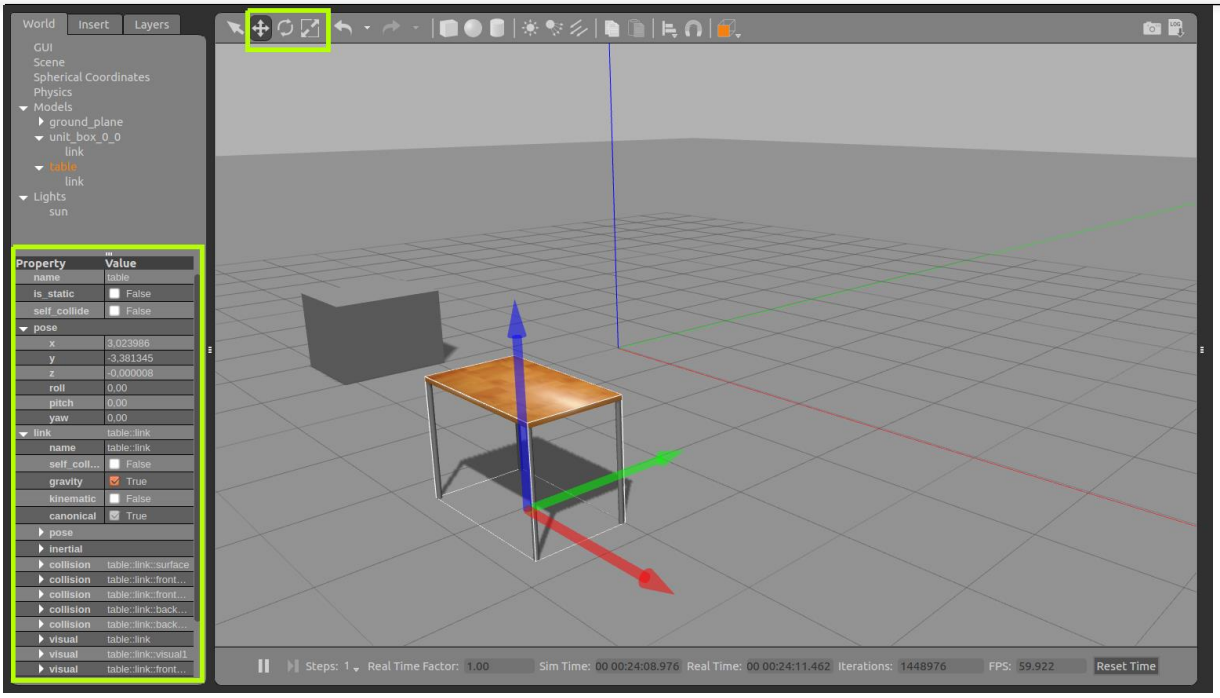

Figure 102 Model Specification

If the world is created as desired, it can be saved from "File" -> "Save World As". In this study, we can save our world in the folder "~ / catkin_ws / src / myrobot / worlds" under the name "first_world_gui.sdf".

After closing Gazebo, we can see the world we created with commands:

```
cd
cd catkin_ws/src/myrobot/worlds/
gazebo first_world_gui.sdf
```

5.3.5. Model Creation

While creating the world for models to be used in the simulation environment, the model file;

- via graphical interface provided "Model Editor"

- manually by proper descriptions compatible with SDF, URDF or URDF.XACRO

Files created using the SDF format can be used directly with Gazebo, while files in URDF and URDF.XACRO format may need extra processing.

To create a model through the "Model Editor" in the graphic interface;

Let's start Gazebo by running the following commands from the terminal in order

```
cd
gazebo
```

Let's open the model editor from the "Edit" -> "Model Editor" section in the toolbar.

When "Model Editor" opens, we have several options to add "Links", that is, a model limb. From the "Insert" section on the left, we can add simple geometric parts to the model with "Simple Shapes", while we can add the "Mesh" parts we have to the model with the "Custom Shapes" option. Gazebo7 supports .svg, .dae and .stl file types as “Mesh” files.

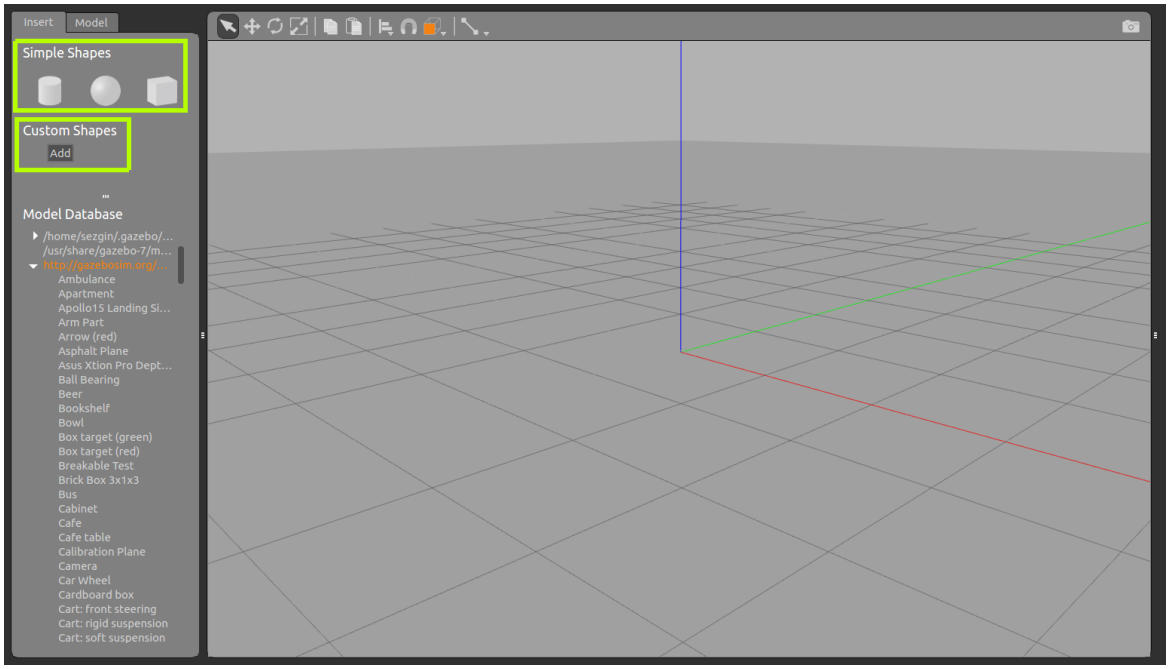

Figure 103 Simple and Custom Shapes

Let's add one "Box" first and then four "Cylinder" to add a total of five "Link" to the model. These limbs will be named as link_0, link_1, link_2, link_3 and link_4 according to the order they were added. Then we can see the "Link" parts we added from the "Model" section on the left. Let's change the model name from the same section, namely "Model Name" to "first_robot_gui".

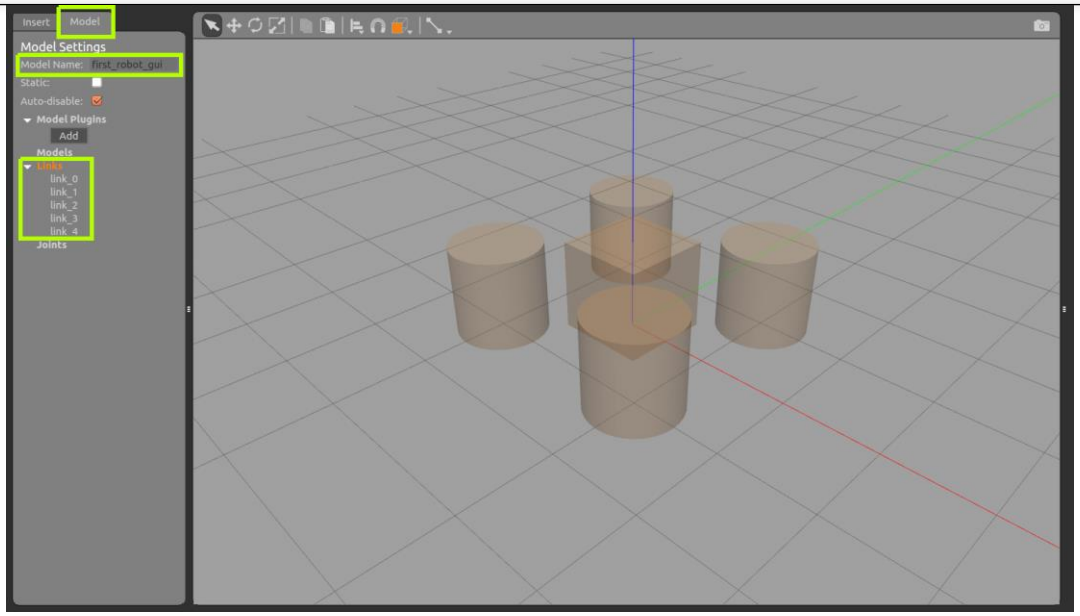

Figure 104 Model Links

In order to change the properties of the parts, first of all, let's click on the "Open Link Inspector" option by right-clicking the "Box" (which will be named link_0 according to the order added).

For link_0, let's set the "Geometry" properties in the "Visual" and "Collision" sections as shown below and get a 1x2x0.5 m "Box" by changing the dimensions of our piece in both visual and solid collision model.

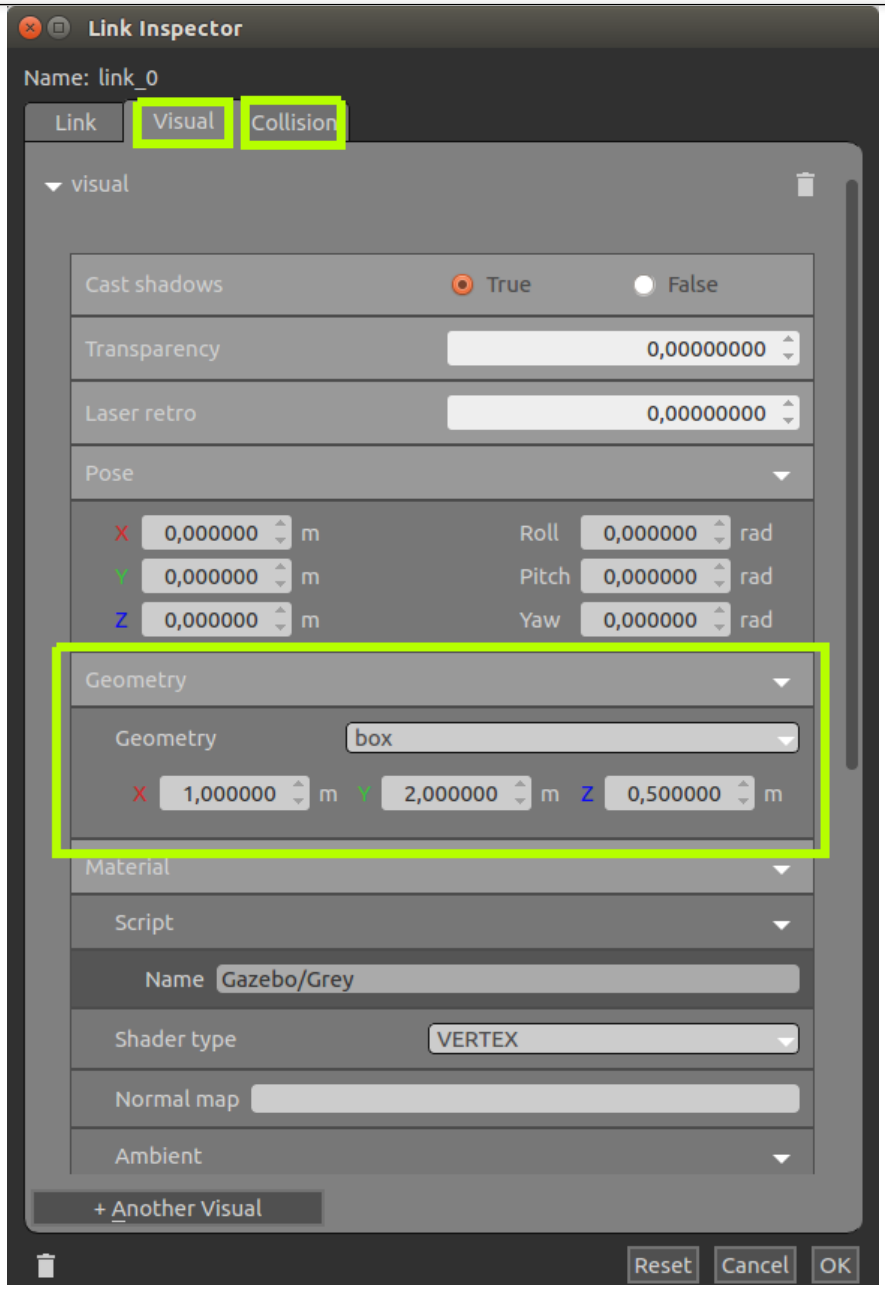

Figure 105 Link Inertia and Geometries

Again, for link_0, this time, from the "Pose" option in the "Link" section, let's set the position of the center point of the "Box" piece to be 0.5m above the z axis according to the origin of the simulation environment as follows. Let's not make any changes to the "Pose" section in the "Inertial" section, since there will be no change in inertial related features.

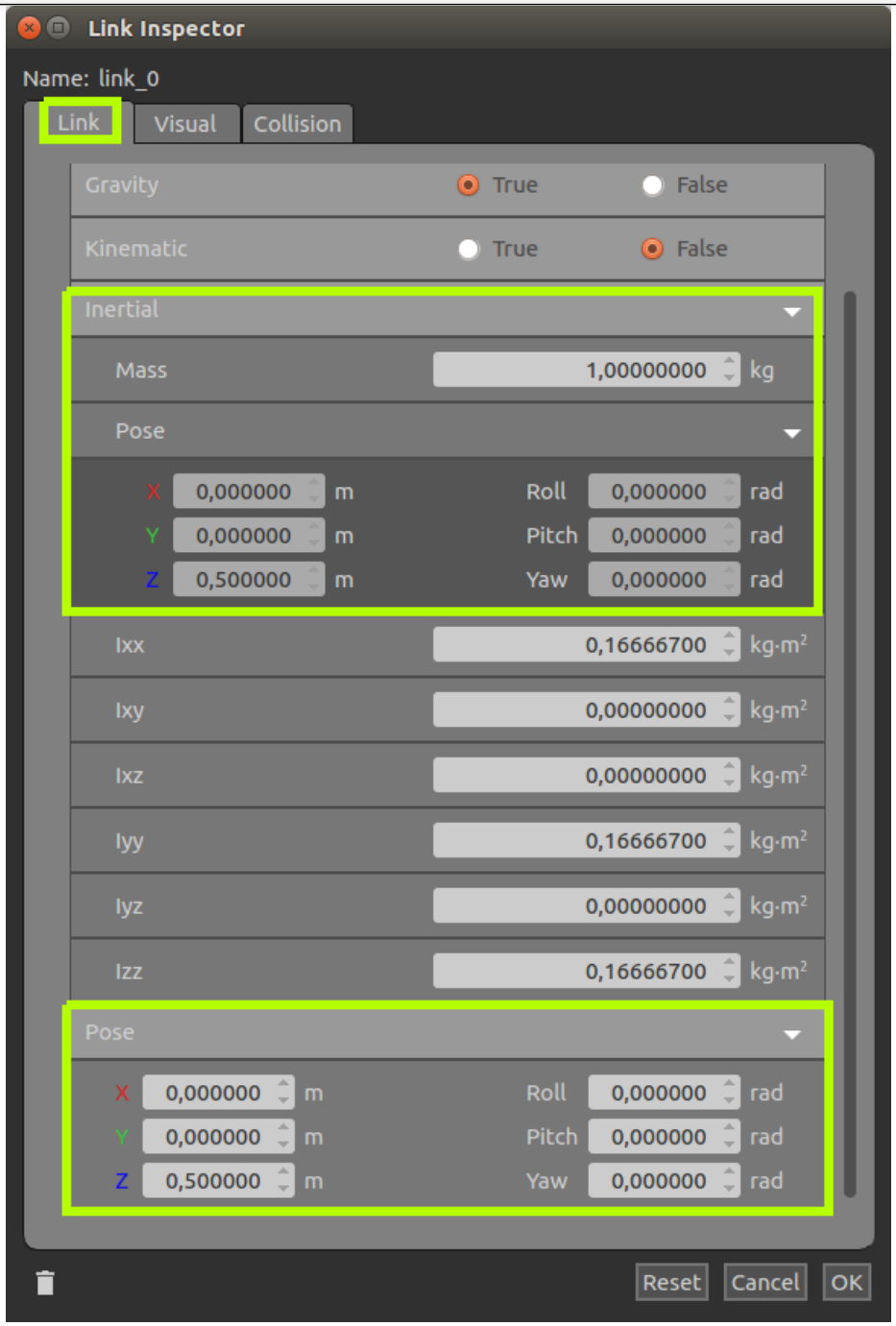

Figure 106 Geometry Specifications

Then, for each "Cylinder" part we added to the environment as link_1, link_2, link_3 and link_4, let's set the dimensions from the "Geometry" field in the "Visual" and "Collision" sections to 0.5x0.5m.

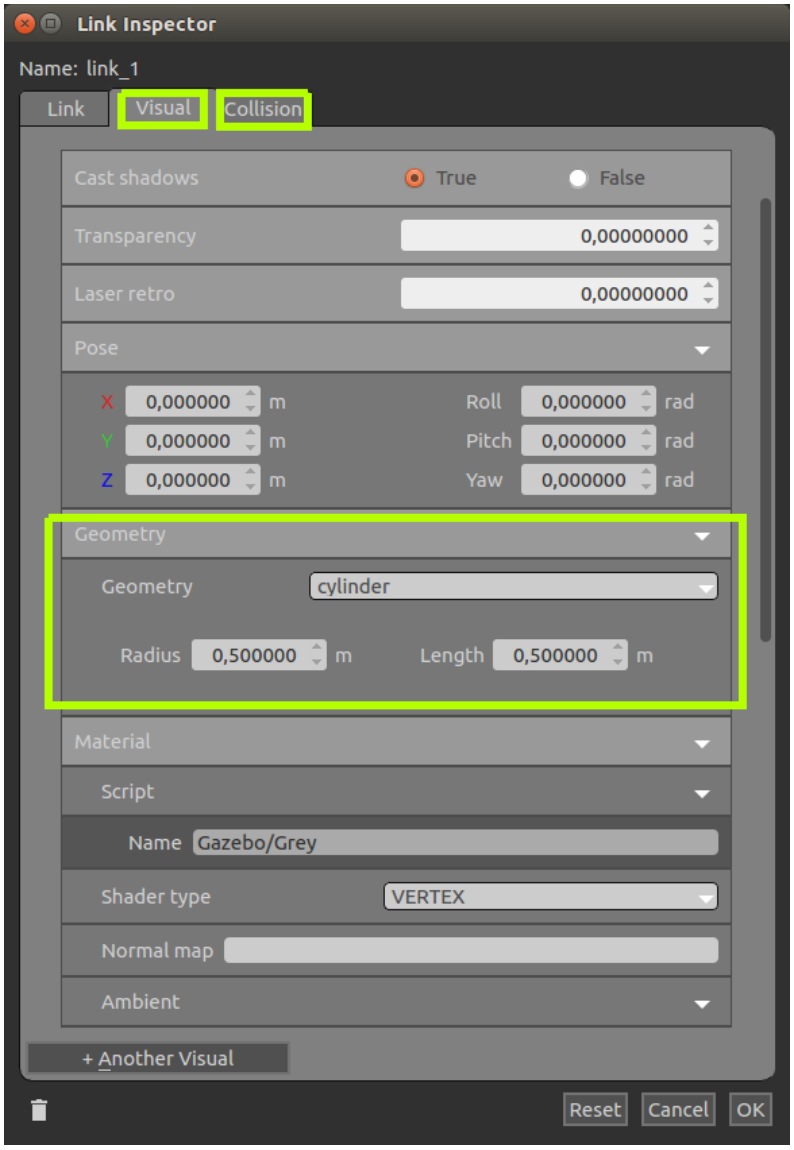

Figure 107 Geometry Area Dimensions

Finally, from the "Pose" option in the "Link" section, let's make the following settings for each link separately.

The values of "Roll", "Pitch" and "Yaw" from the "Pose" section in the "Pose" sections of each will be "0.000000", "1.570000" and "0.000000" respectively. "X", "Y" and "Z" values in the "Pose" values in the "Pose" sections;

- For link_1;
  - X : -0,750000
  - Y : -1,000000
  - Z: 0,500000

- For link_2;
  - X : -0,750000
  - Y : 1,000000
  - Z: 0,500000

- For link_3;
  - X : 0,750000
  - Y : 1,000000
  - Z: 0,500000

- For link_4;
  - X : 0,750000
  - Y : -1,000000
  - Z: 0,500000

will be settled. When you set “Link” parts correctly, the image created in the simulation will be as shown below.

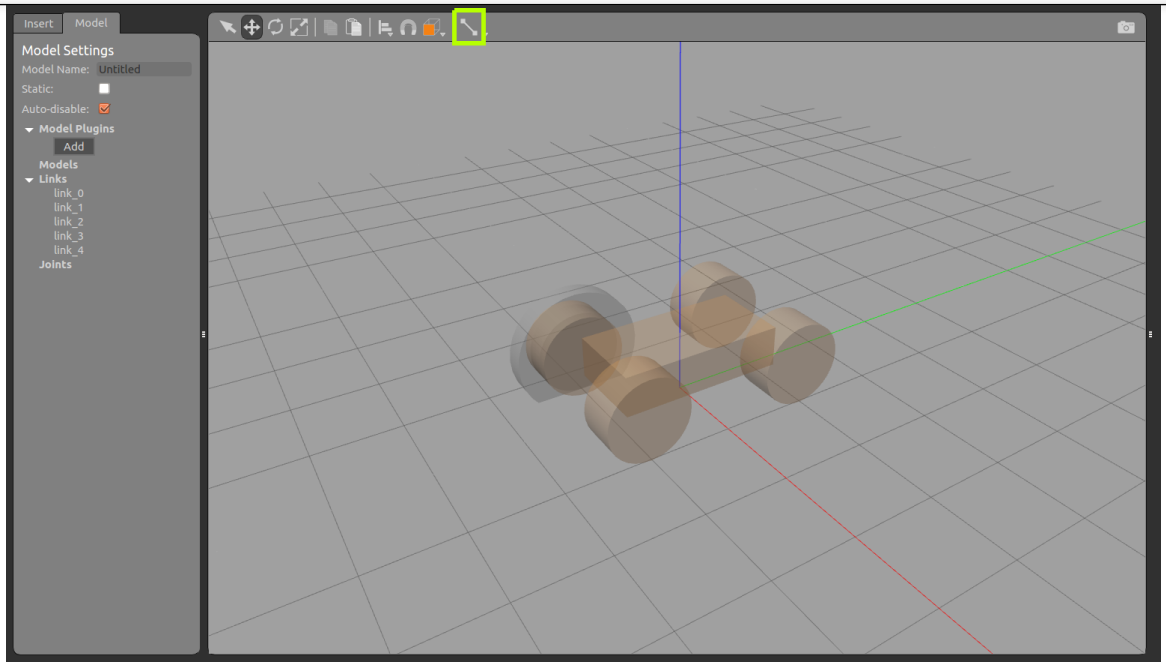

Figure 108 Created Model

To add “Joints” to our model, that is, we can click the “Joint” button from the toolbar at the top as shown in the top figure. Here we can select the type of joint, the choice of upper and lower links, and the joint axis. After making all the settings, we can create a "Joint" with the "Create" option. Since we will do this for each sub link, we have to create four “Joints” in this study.

- "Joint types" "Revolute" for each joint,
- “Parent” in “Link Selections” section is “link_0” for each joint,
- "Link_x" (x 1,2,3,4 respectively) for the "Child" related joint in the "Link Selections" section,
- "Joint axis" "Z" for each joint

Let's go to the settings of another joint by setting "Create" after each setting..

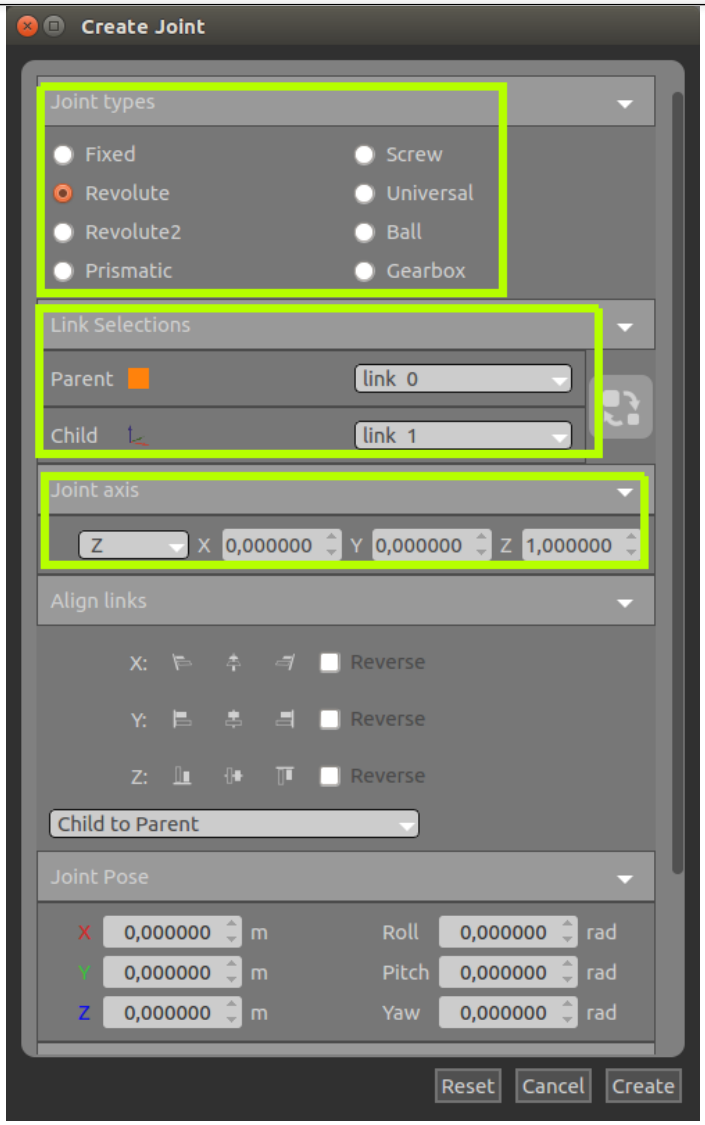

Figure 109 Creating Joints

When we make the “Joints” settings correctly, the image created in the simulation will be as shown below.

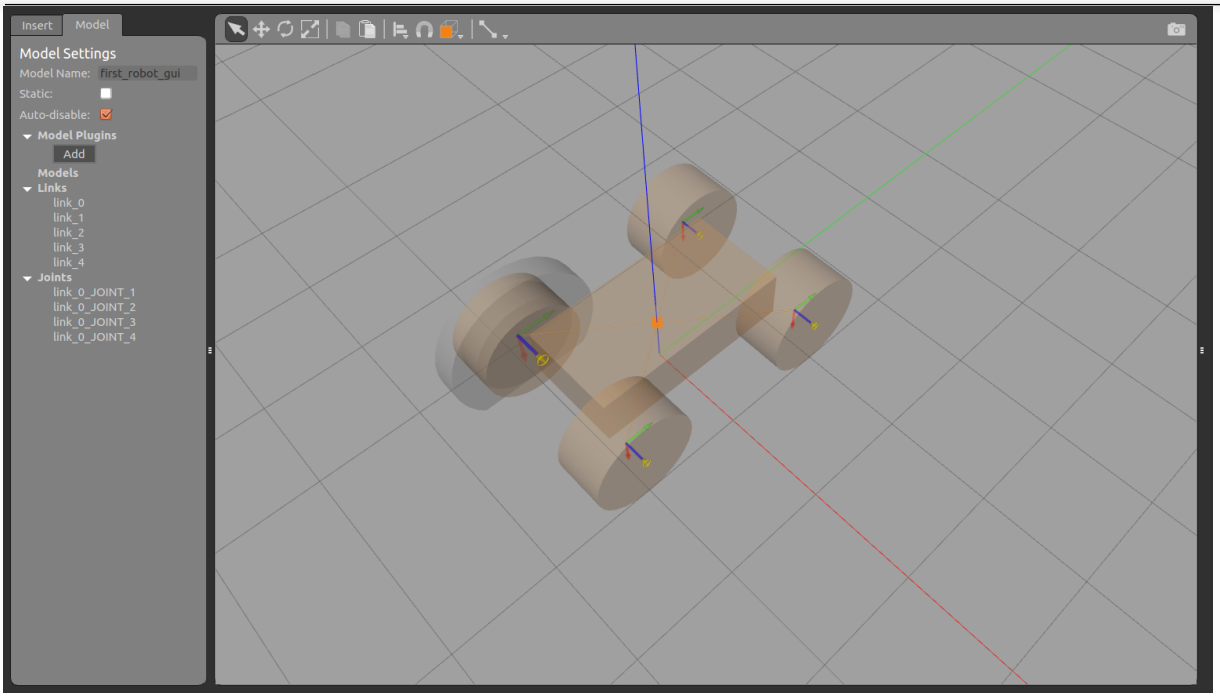

Figure 110 Created Model

f we were able to create the model correctly, we can save our model. The registration process will create a directory, SDF file and configuration file for our model. In order to save, let's choose “File” -> “Save As”. Registration options;

Model Name: first_robot_gui
Location: .gazebo / models / first_robot_gui

Let's set it and save it. Next, let's exit the "Model Editor" interface, using "File" -> "Exit Model Editor". The robot we created will be added to the simulation environment.

5.3.6. Importing Mesh

While creating the models, a model file can be defined on the <visual> tag and the model can be printed on the model.
If the <visual> tag is defined, only the visual feature of the model is affected.

For this, the "Visuals" tag of the "Chassis" link in the section below;

```
<!--visual>
    <origin xyz="0.0 0.0 0.1" rpy="0.0 0.0 0.0" />
    <geometry>
        <box size="0.40 0.20 0.10"/>
    </geometry>
</visual-->
```

By changing it to "comment", let's add the following part below;

```
<visual>
        <origin xyz="0.0 0.0 0.2" rpy="0.0 0.0 0.0"
/>
        <geometry>
            <mesh
filename="model://pioneer2dx/meshes/chassis.dae"
scale="1.1 0.75 1.5"/>
        </geometry>
</visual>
```

The first_robot.urdf.xacro model we updated;

```
cd
    cd ~/catkin_ws
    roslaunch myrobot first_robot.launch
```

We can see it on Gazebo with commands.

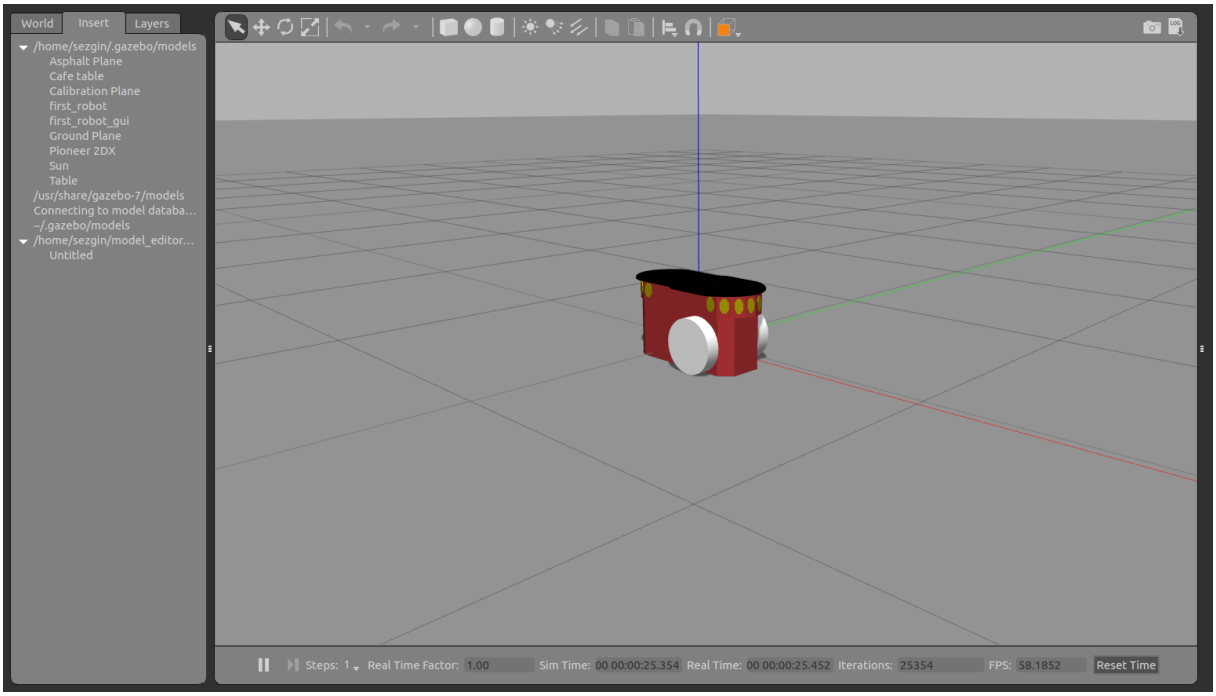

Figure 111 Fitted Mesh

5.3.7. Adding Sensors

While creating models, a sensor can be added to the model by defining <link>, <joint> and if necessary, <gazebo reference> for the model file for .urdf files.

To add a laser sensor to the first_robot.urdf.xacro model we created;

```
gedit
~/catkin_ws/src/myrobot/urdf/first_robot.urdf.xacro
```

Let's add sensors by updating our model with commands.

For this, before adding the "</robot>" tag at the end of the file, add the following part;

```
    <link name="laser">
            <collision>
                    <origin xyz="0 0 0" rpy="0 0 0"/>
                    <geometry>
                    <box size="0.1 0.1 0.1"/>
                    </geometry>
            </collision>

            <visual>
                    <origin xyz="0 0 0" rpy="0 0 0"/>
                    <geometry>
                    <mesh
    filename="model://hokuyo/meshes/hokuyo.dae"/>
                    </geometry>
            </visual>

            <inertial>
                    <mass value="1e-5" />
                    <origin xyz="0 0 0" rpy="0 0 0"/>
                    <inertia ixx="1e-6" ixy="0" ixz="0"
    iyy="1e-6" iyz="0" izz="1e-6" />
            </inertial>
         </link>


        <joint name="hokuyo_joint" type="fixed">
            <axis xyz="0 0 0" />
            <origin xyz="0.2 0 0.3" rpy="0 0 0"/>
            <parent link="chassis"/>
            <child link="laser"/>
        </joint>
```

```
    <gazebo reference="laser">
        <sensor type="gpu_ray"
name="head_hokuyo_sensor">
            <pose>0 0 0 0 0 0</pose>
            <visualize>true</visualize>
            <update_rate>40</update_rate>
            <ray>
            <scan>
                    <horizontal>
                        <samples>720</samples>
                        <resolution>1</resolution>
                        <min_angle>-
3.1416</min_angle>

    <max_angle>3.1416</max_angle>
                    </horizontal>
            </scan>
            <range>
                    <min>0.5</min>
                    <max>5.0</max>
                    <resolution>0.1</resolution>
            </range>
            <noise>
                    <type>gaussian</type>
                <mean>0.0</mean>
                    <stddev>0.01</stddev>
            </noise>
            </ray>
```

```
            <plugin
    name="gazebo_ros_head_hokuyo_controller"
    filename="libgazebo_ros_gpu_laser.so">

        <robotNamespace>first_robot</robotNamespace>
                        <topicName>sensor/Laser</topicName>
                    <frameName>laser</frameName>
                    </plugin>
                </sensor>
      </gazebo>
```

The first_robot.urdf.xacro model we updated;

```
    cd
    cd ~/catkin_ws
    roslaunch myrobot first_robot.launch
```

We can see it on Gazebo with commands.

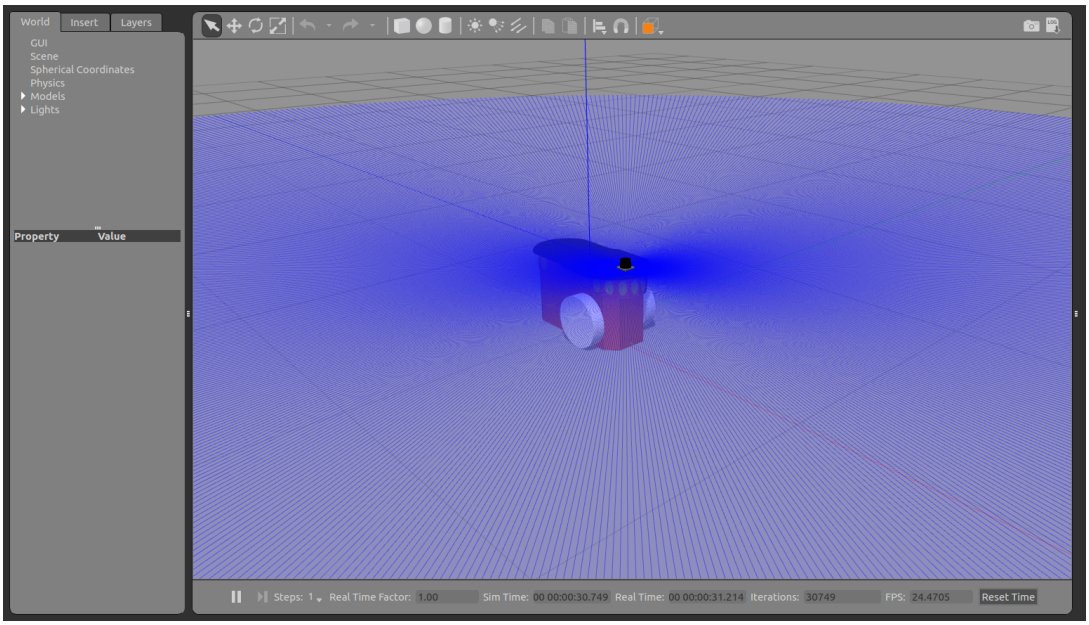

Figure 112 Sensor visualization

5.4. Control with ROS

In order to control the model we created with ROS in the simulation environment, the related packages must first be installed. For example, for ROS Kinetic;

```
sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-ros-control
```

About the command gazebo_ros_pkgs can be downloaded. Then a catkin_workspace environment should be prepared as done at the beginning of this study.

If the ROS and Gazebo environment is properly created and configured

```
rosrun gazebo_ros gazebo
```

Gazebo will run the command successfully.

As already shown in earlier parts of the study

```
roslaunch myrobot first_robot.launch
```

The launch of our robot from the launch file we created with the command is also among the things to be done for control with ROS.

6. MoveIt

6.1. What is MoveIt?

MoveIt, widely used for robot manipulation;

- developing advanced applications,
- evaluating new designs,
- creating integrated products

MoveIt provides an easy-to-use robotic platform.

6.2. MoveIt Content

Thanks to the facilities it contains as MoveIt platform;

- Motion planning
- robot manipulation

- tnverse kinematic analysis
- robot control
- 3-D sensing
- Collision control

Applications can be realized.

With the plugin named “Rviz Motion Planning Plugin”, it is possible to try various planning algorithms in environments where there are obstacles.

Along with various planning libraries such as "OMPL", "CHOMP" and "STOMP", it is possible to use the current planning algorithms in the literature.

With the configuration wizard called “MoveIt Setup Assistant”, it is possible to configure any robot step by step or to use popular pre-configured structures.

By integrating Gazebo, ROS Control and MoveIt, a powerful robot development platform can be obtained.

6.2.1. MoveIt Installation

If ROS is successfully installed and MoveIt is not yet installed, installation can be done using the following command using pre-built binaries for Kinetic;

```
sudo apt install ros-kinetic-moveit
```

After this stage, catkin_workspace can be created. The workspace created earlier in this work can also be used for this work. Let's run the following commands in order to set the current working environment;

```
cd ~/catkin_ws/src/
mkdir myrobot_moveit_config
cd myrobot_moveit_config/
```

6.2.2. MoveIt Setup Assistant

"MoveIt Setup Assistant" is a user interface used to configure any robot to be used with MoveIt. As a result of using this interface, the Semantic Robot Description Format (SRDF) file and other necessary configuration files are created to be used in the MoveIt pipeline.

In order to create the configuration package of a robot, let's start the "MoveIt Setup Assistant" first;

```
roslaunch moveit_setup_assistant
setup_assistant.launch
```

and then click "Create New MoveIt Configuration Package" button.

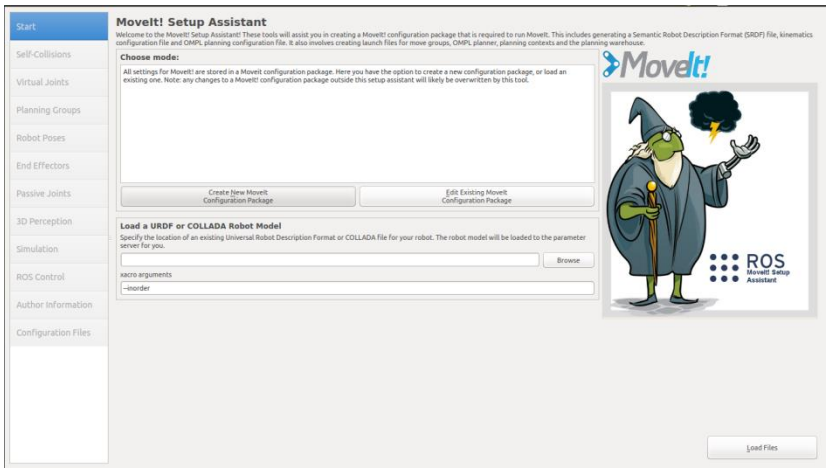

Figure 113 MoveIt Interface

Let's select the .urdf or .urdf.xacro file of our robot on the page that opens and click on the "Load Files" button. After these processes, our robot will appear on the right side of the interface. On the left side, there will be the steps we will follow for the configuration we will perform.

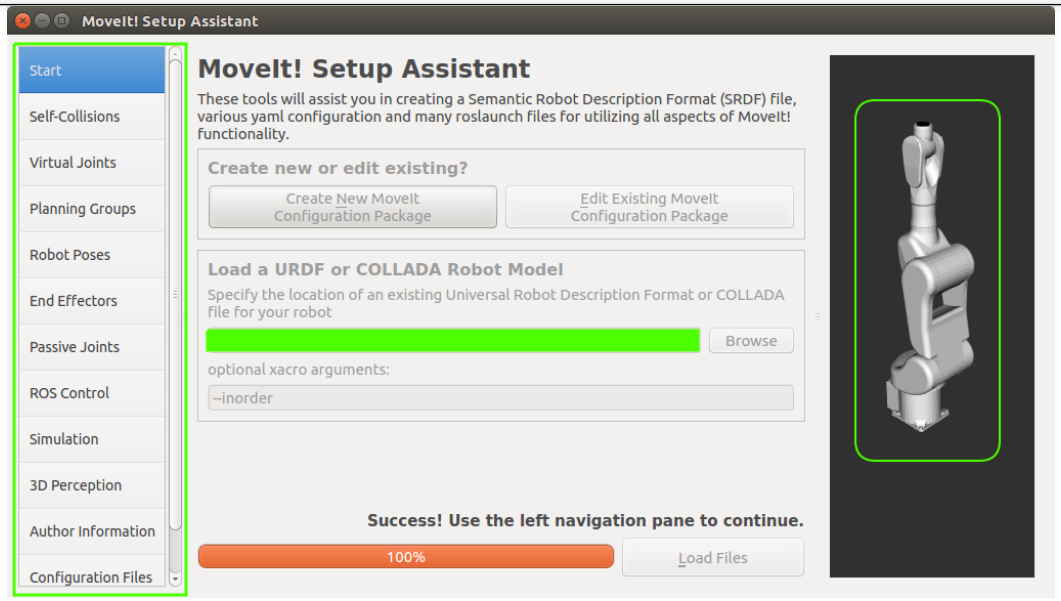

Figure 114 MoveIt Setup assistant

Let's first click on the “Self-Collisions” option for the configuration. On the page that opens, self-collision control will be optimized. This process aims to reduce motion planning time. To perform, let's set the “Sampling Density” as desired and click on the “Generate Collision Checking” option. "Sampling Density" shows how many random robot positions are determined for self-collision control. If this value is kept high, the processing time to be performed at this stage may be longer, but the results obtained may be more sensitive.

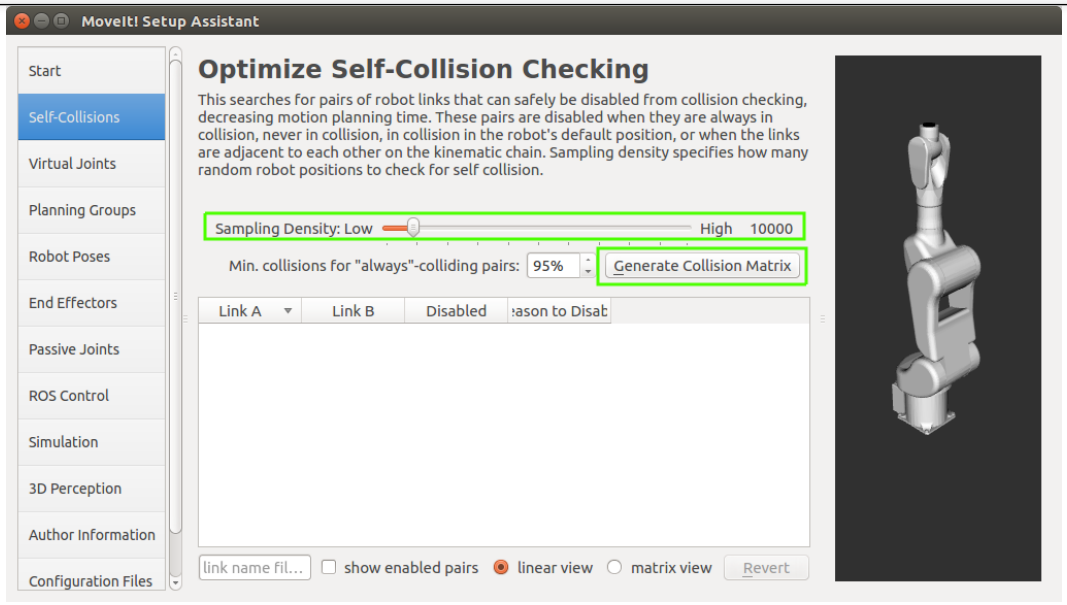

Figure 115 Self-collusion Checking

After generating the collision matrix, robot link pairs or adjacent robot link pairs that will never be in collision as below will be identified.

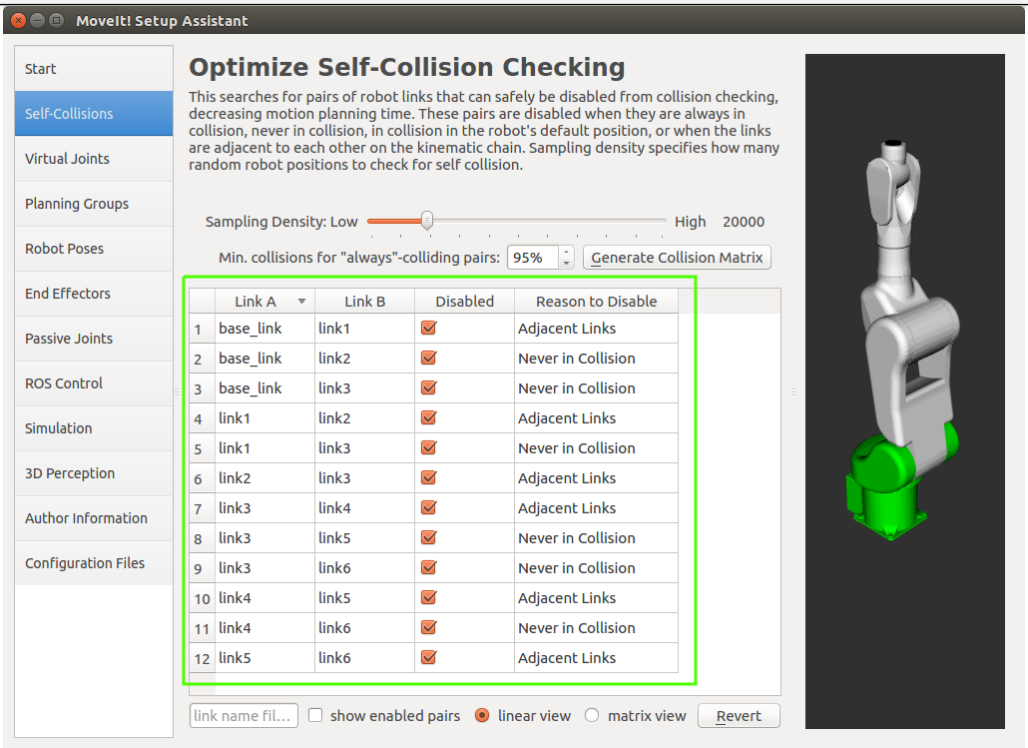

Figure 116 Robot Link Couples

In the next step, we can define a virtual joint to connect the robot to the world by clicking on "Virtual Joints". To do this, after clicking “Virtual Joints” option, let's click “Add Virtual Joint” option and create a virtual joint as below.

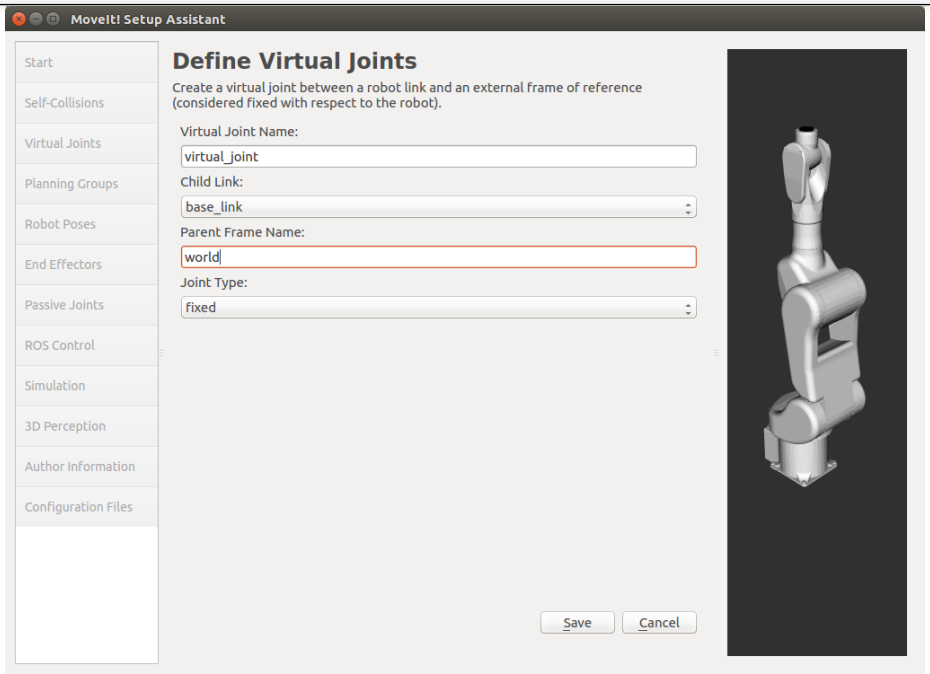

Figure 117 Virtual Joint Creation

For this, the following options;

Virtual Joint Name: virtual_joint
Child Link: base_link
Parent Frame Name: world
Joint Type: Fixed

Can be determined.

In the next step, "Planning Group" can be determined. “Planning Group” is used to semantically define different parts of the robot, such as what is a robot arm or defining the end point of the robot. To do this, after clicking on the "Planning Group" option, let's click on the "Add Group" option and create a planning group as follows.

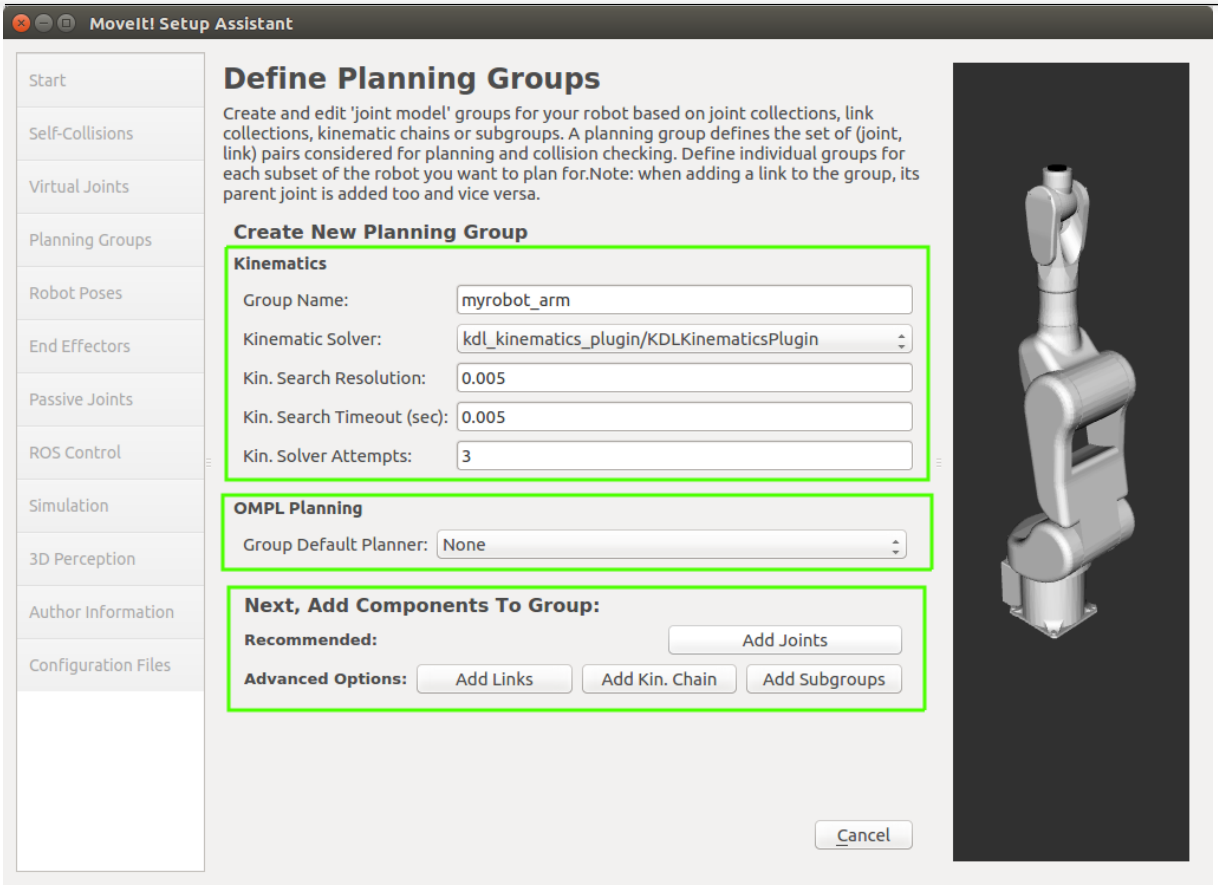

Figure 118 Defining Planning Groups

“Kinematics” part as follows;

Group Name: myrobot_arm
Kinematic Solver: kdl_kinematics_plugin / KDLKinematicsPlugin
Kin.Search Resolution: 0.005
Kin.Search Timeout (sec): 0.005
Hatred. Solver Attempts: 3

can be described. The group name is used to identify the group in which planning will be performed. The KDL kinematic plug-in uses the numerical inverse kinematics solver provided by the Orocos KDL package. This plugin is the default kinematic plugin currently used by MoveIt. In addition, if a powerful custom inverse kinematics solver is desired, the IKFast MoveIt plugin can be created by the user.

In the "OMPL Planning" section, the desired motion planning algorithm can be selected or left unselected for the default planner.

In the "Next, Add Components to Group" section, we can click on "Add Joints" and add the joints as below and click on the "Save" option.

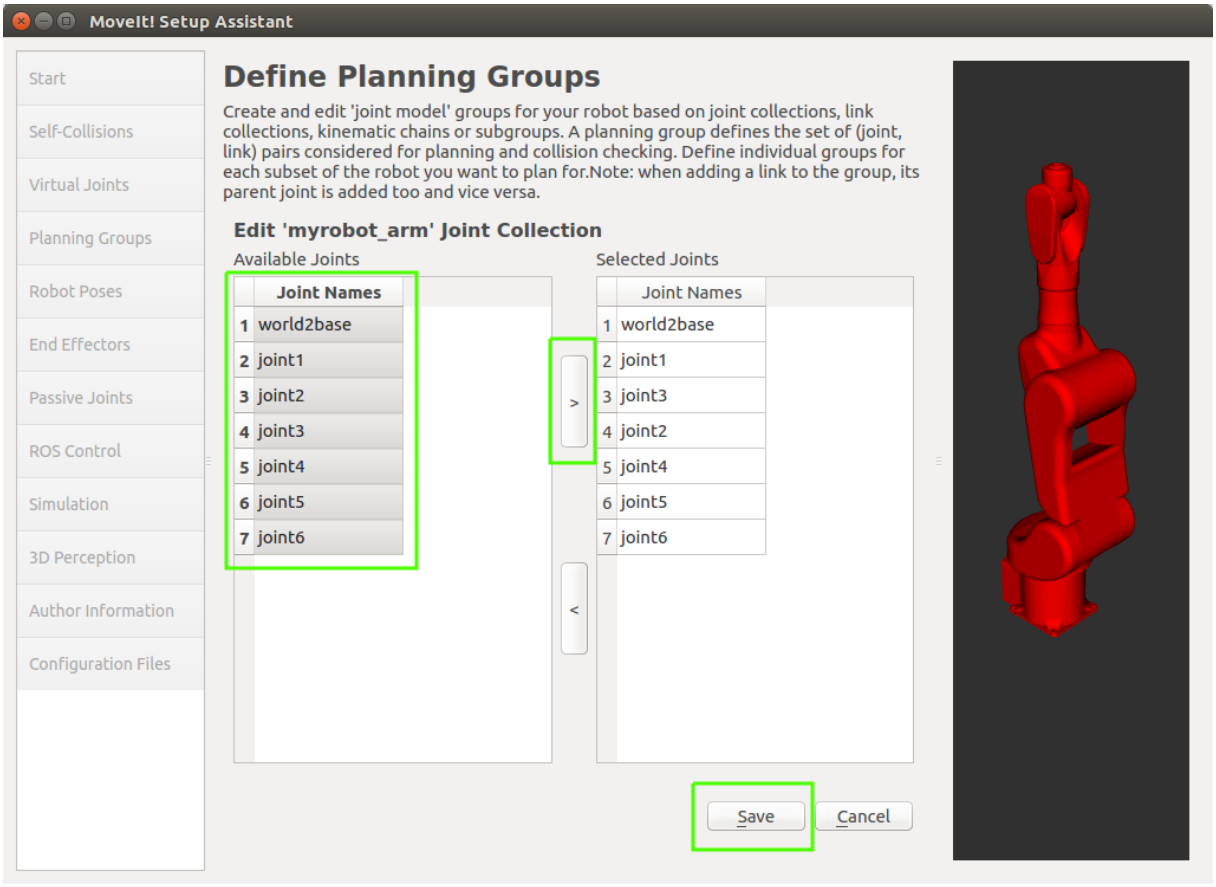

Figure 119 Defining and adding joints

On the screen after selecting the joints, we can set the "Link" option and then the "Chain" option as follows. For this, we can select the relevant option and click the "Edit Selected" button.

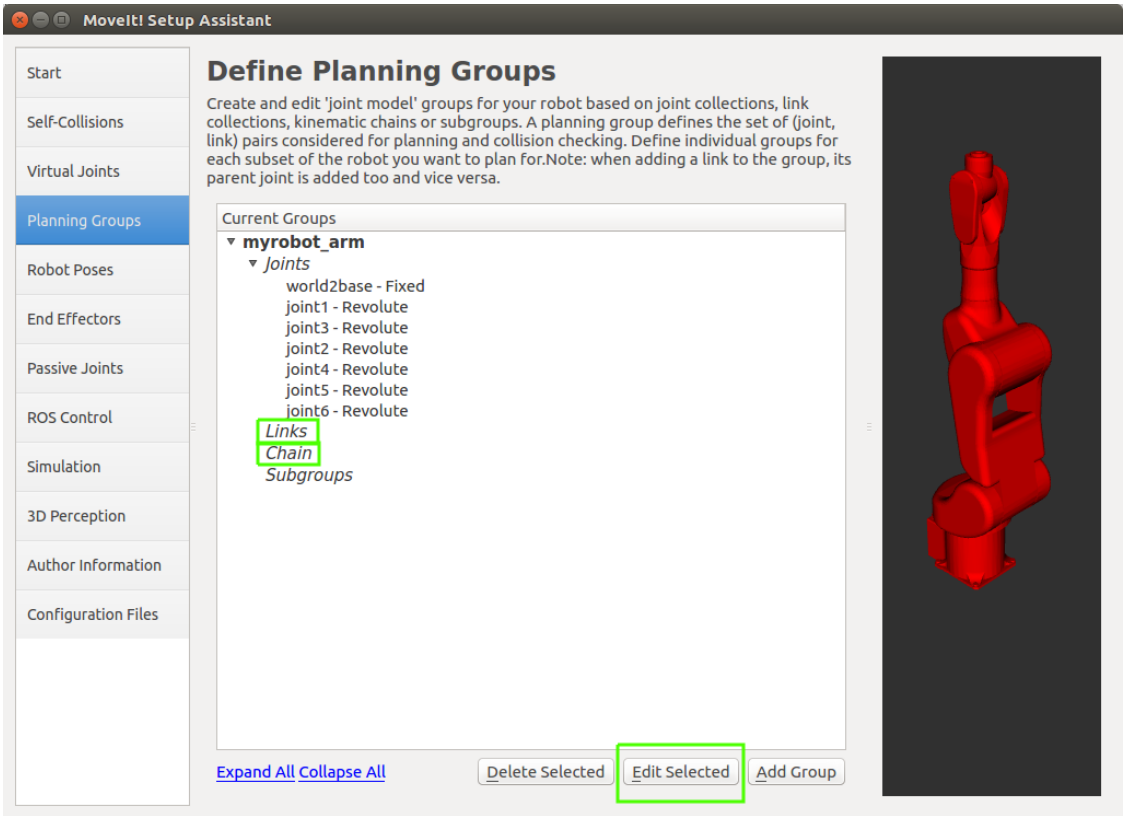

Figure 120 Editing Selected Groups

While selecting the robot limbs as below for the "Link" option;

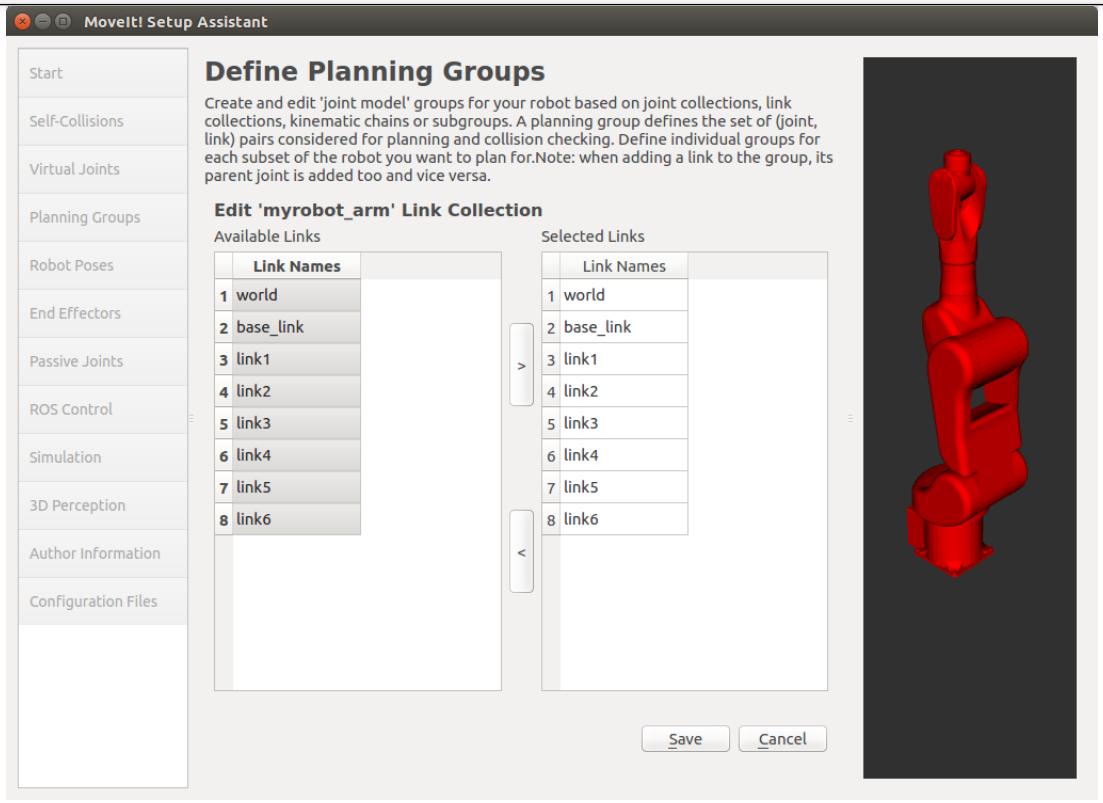

Figure 121 Robot Link Collection

In the "Chain" option, we can define the starting limb and the end effector in the kinematic chain of the robot as follows;

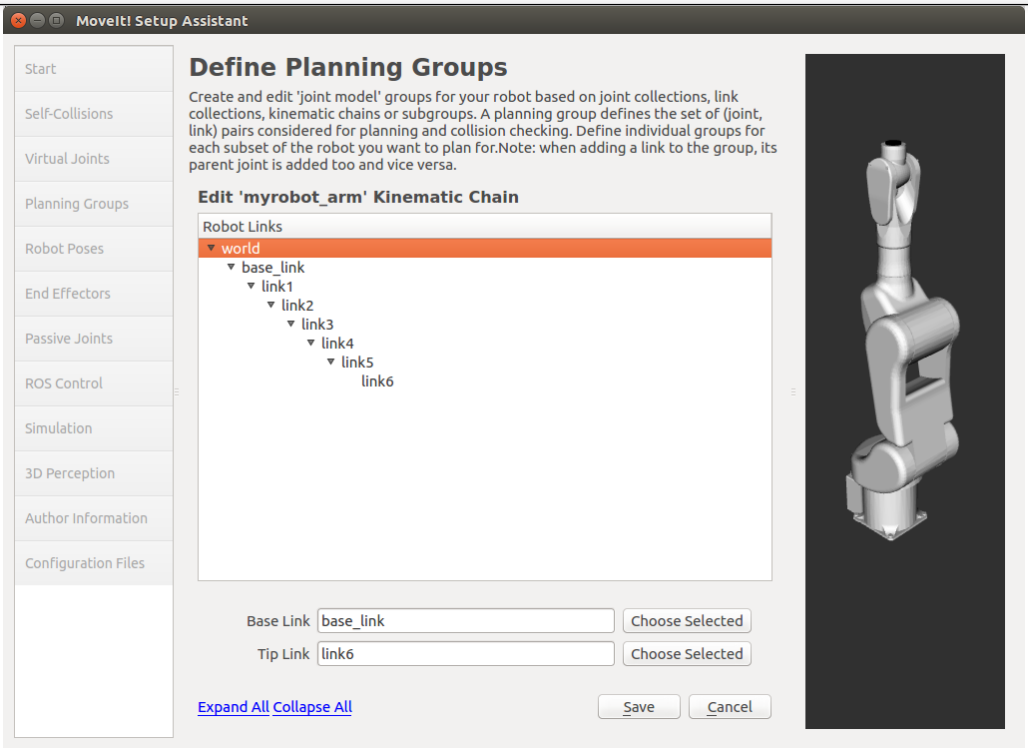

Figure 122 End Effector Definiton

Finally, we get a configuration for the planning group as follows.

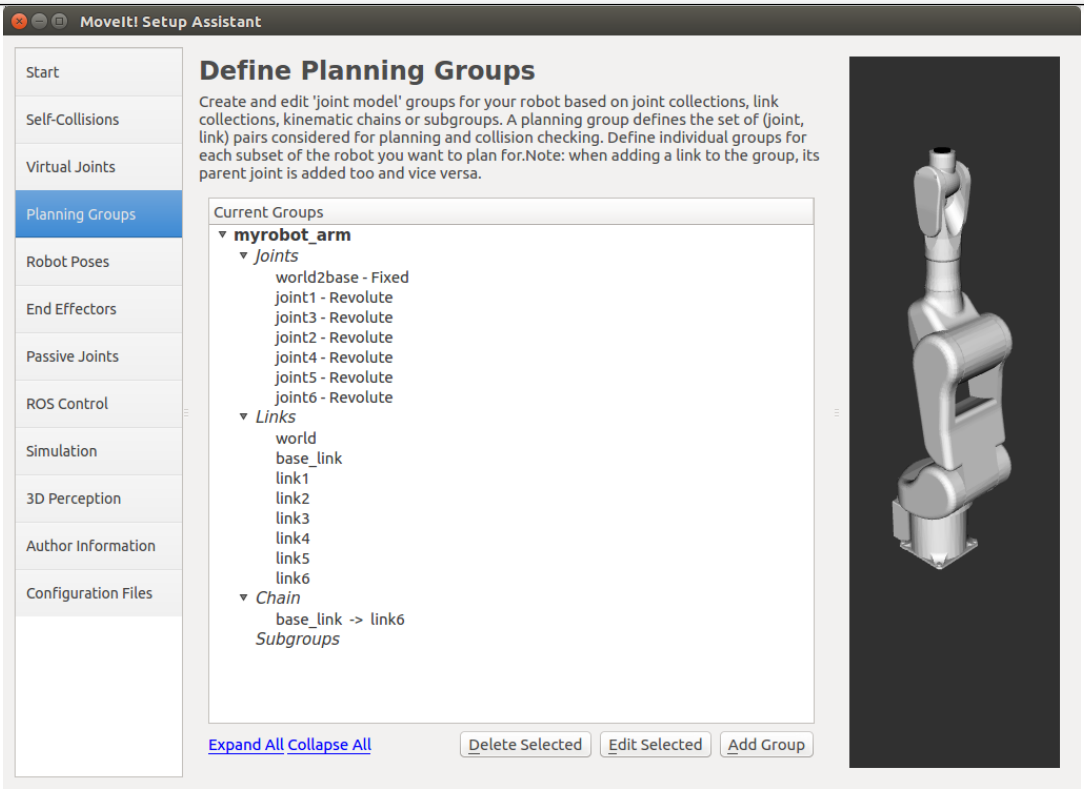

Figure 123 Created Robot Modification

Similarly, we can define another group at the robot tip that we can define as "tool" and we can create another group called "tool" as below.

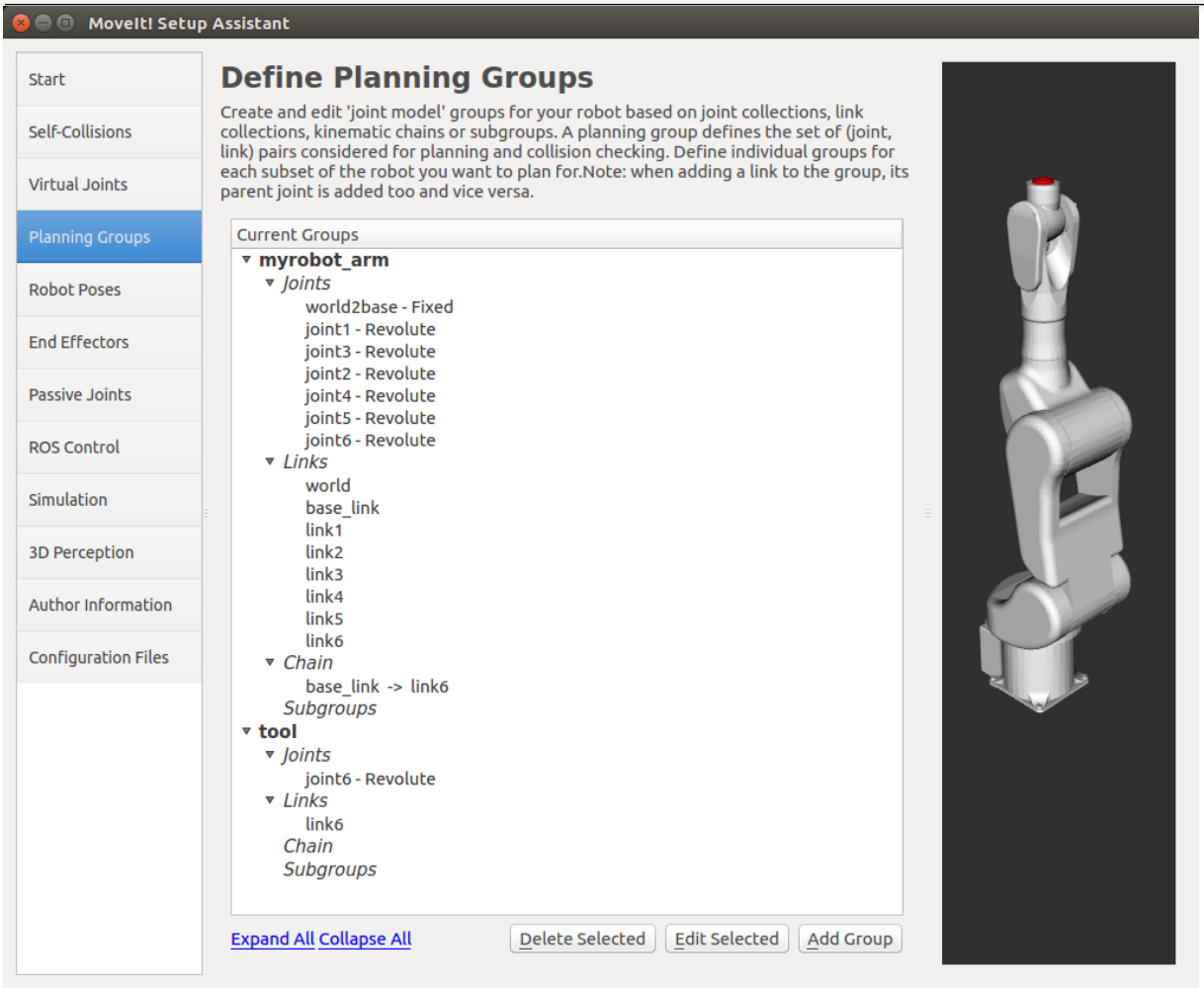

Figure 124 Tool Creation

After that, it is also possible to add certain fixed robot stops to the configuration from the "Robot Poses" section. For example, we can define a first stance for the robot that we can refer to as "home". To do this, click on the "Add Pose" option as follows;

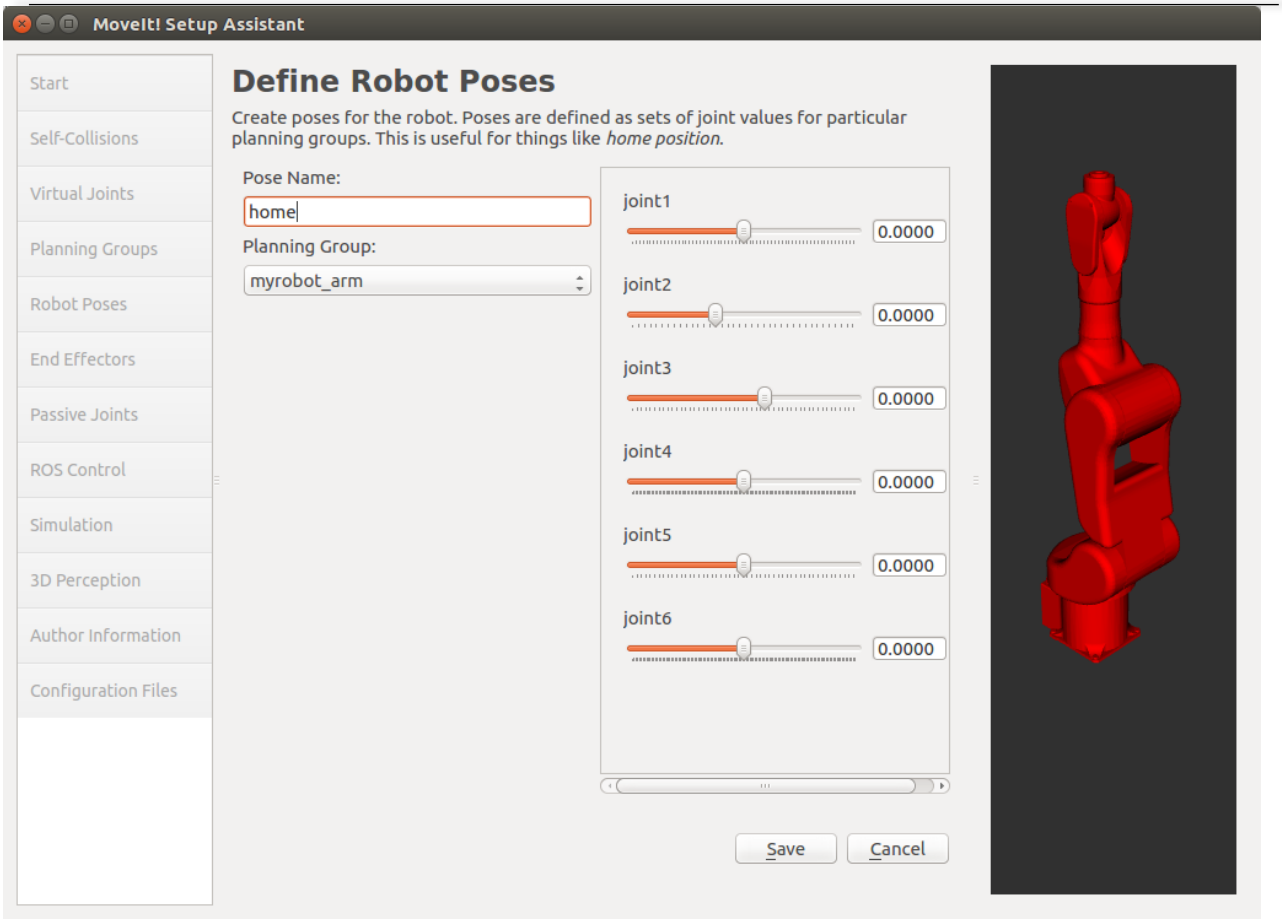

Figure 125 Defining Robot Positions

We can define a posture that we can determine, a posture by filling the relevant "Pose Name" as "home" and clicking the "Save" button.

In the next step, we can determine the group that we designated as the "tool" in the previous two stages from the "End Effectors" section as a special end point group. To do this, by clicking on the "Add End Effector" option;

End Effector Name: ee
End Effector Group: tool
Parent Link: link6

We can define the end point in the form.

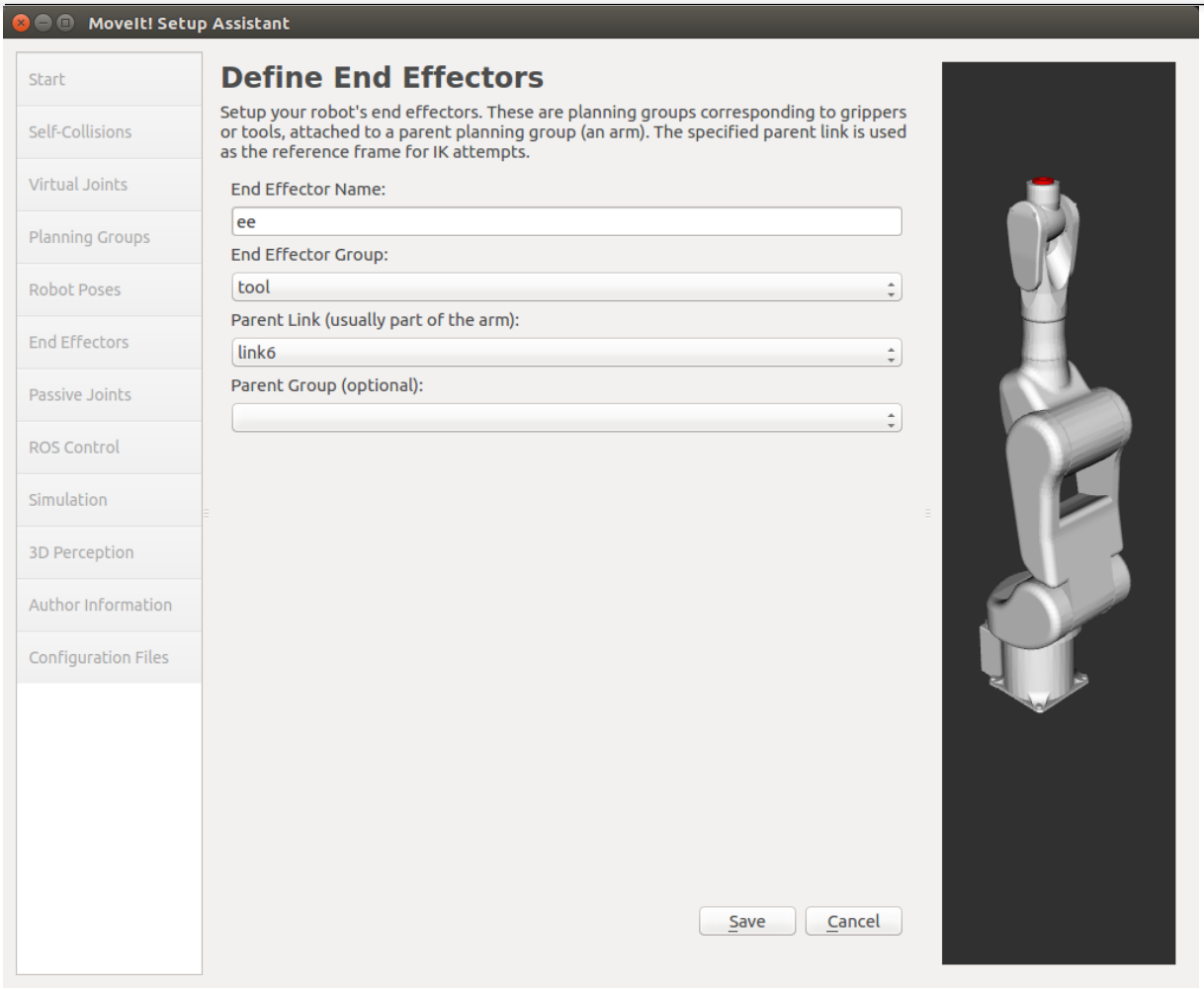

Figure 126 Defining End Effectors

The “Passive Joints” section is used to identify passive joints that can be found in a robot. In this way, since passive joints cannot be controlled directly (kinematically), there will be no need to plan for these joints. Since there is no such joint in the robot model used in this study, no action is required.

In the “ROS Control” section, simulated controllers can be created automatically to activate the joints of the robot. In this section, clicking on “Auto Add FollowJointsTrajectory Controllers For Each Planning Group” can be done as follows. When you want to choose another type as the controller type, the type can be changed by clicking on the relevant controller and clicking on “Edit Selected”.

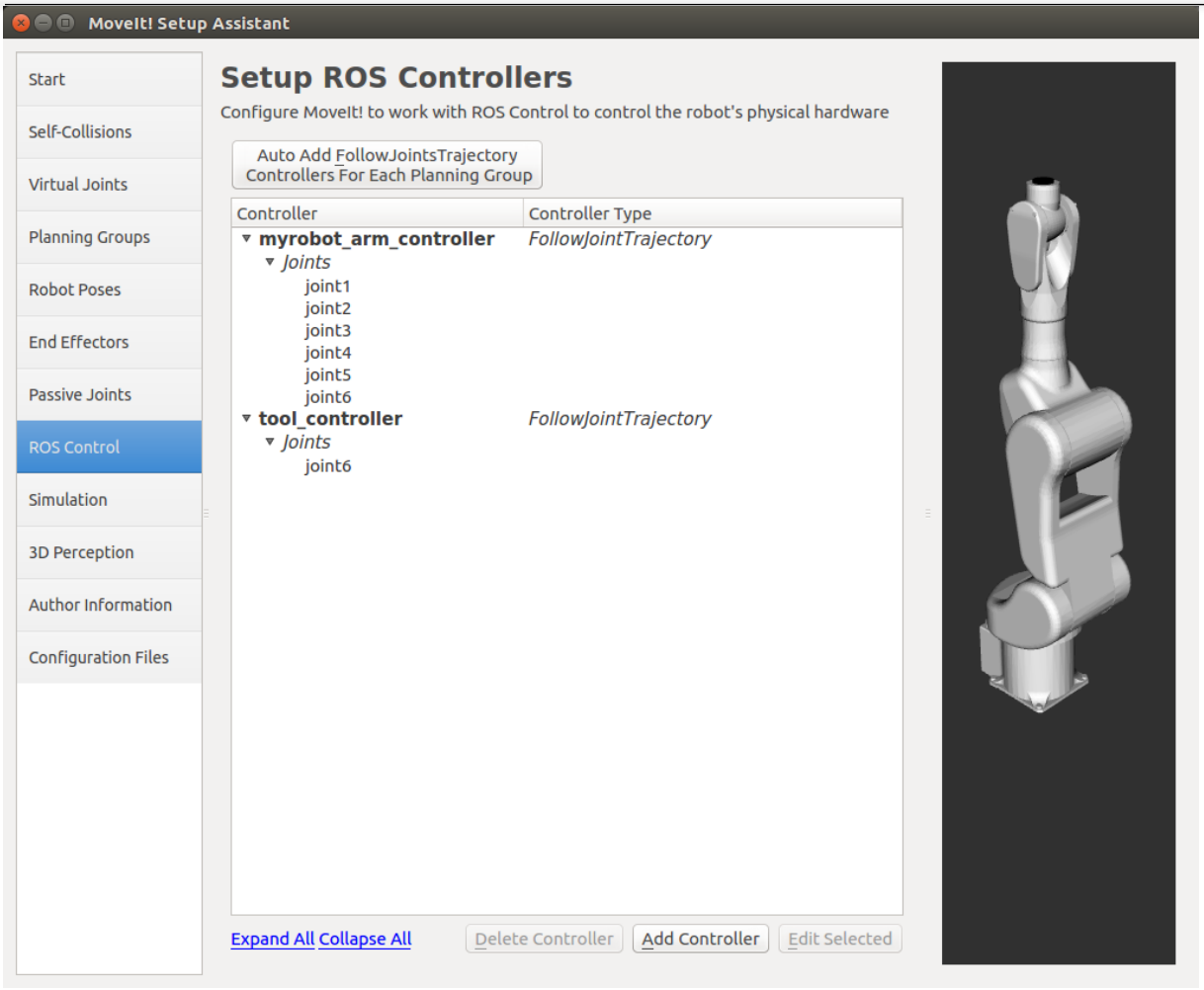

Figure 127 Control With ROS

In the "Simulation" section, changes required in the URDF file are automatically generated in order for ROS Control and MoveIt to work in accordance with Gazebo. The changes that need to be made are shown in green. If desired, these changes can be added to the relevant URDF file by copy.

In the "3D Perception" part, it is possible to configure the 3D sensors by adjusting the parameters of the YAML configuration file. Since there is no such sensor in the robot model used in this study, no action is required.

In the "Author Information" section, information about the person who created the package can be entered.

Finally, in the "Configuration Files" section, we can complete the configuration. For this, we can choose the path where the configuration will be saved as below and click on the “Generate Package” option.

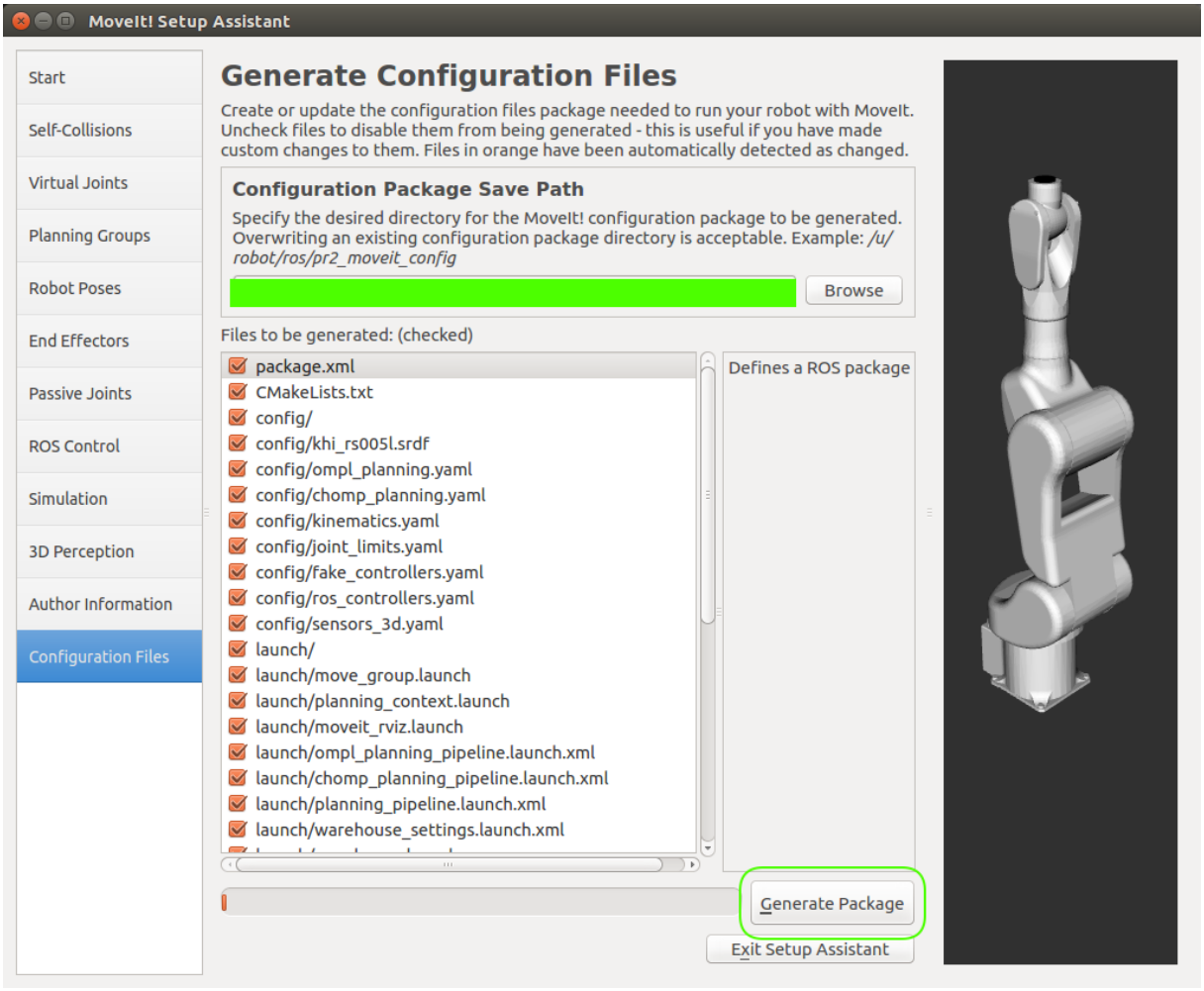

Figure 128 Creating Packages

At the end of these stages, configuration files are created to be used for planning with MoveIt.

6.2.3. MoveIt Planning and Demonstration of Work on Gazebo with Rviz

When the MoveIt package was created correctly, it revealed the robot both on Rviz and on Gazebo;

Hareket Motion planning can be done with the “Rviz Motion Planning Plugin” plugin.
Durumu By running the created plan, the status of the robot can be observed on Gazebo.

When the MoveIt package is created correctly;

```
roslaunch myrobot_moveit_config demo_gazebo.launch
```

With the "Rviz Motion Planning Plugin" plugin, we can detect the robot on both Rviz and Gazebo, and run the plan and observe the robot's status on Gazebo (We can also define separate launch files to perform this operation).

As a result of these transactions, Rviz is as follows;

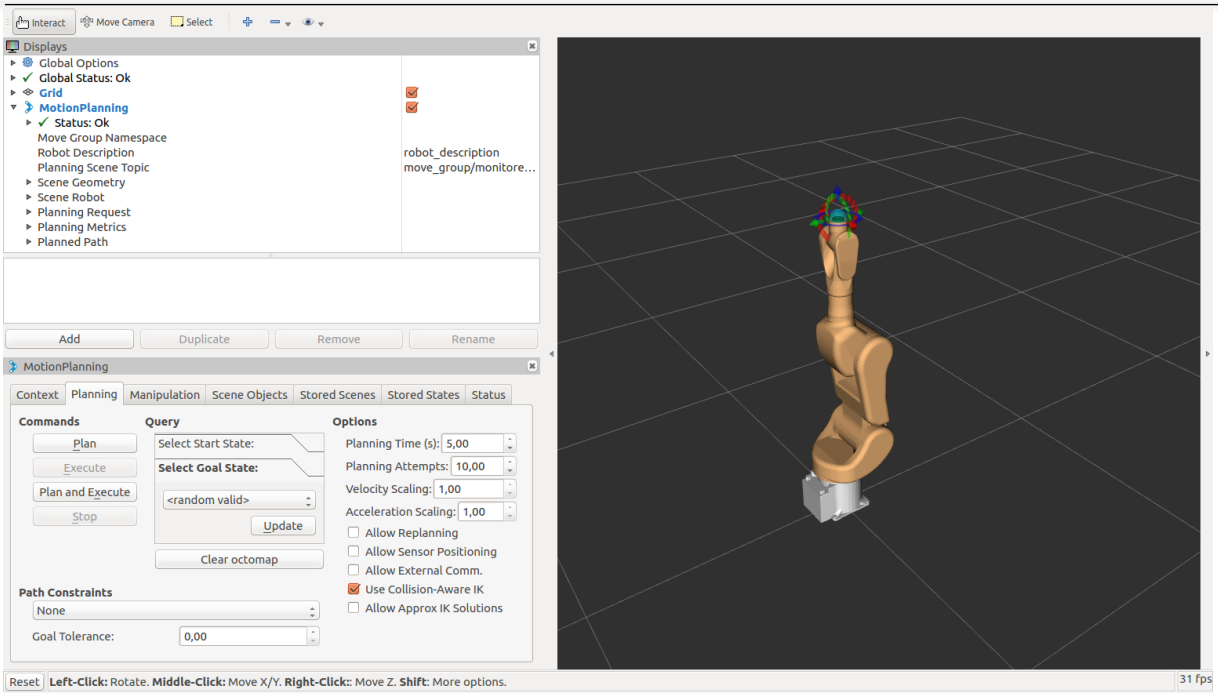

Figure 129 Rviz interface

On the Gazebo, as follows;

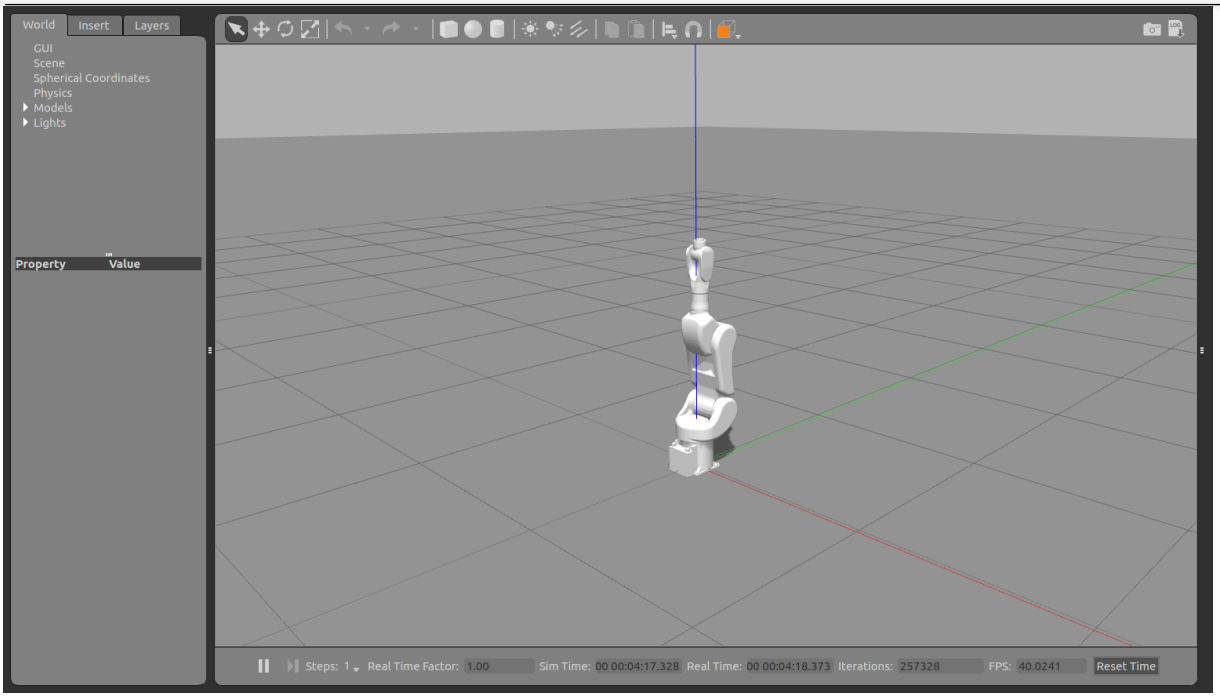

Figure 130 Gazebo interface

we get a situation.

For a sample action planning, we can first select the "Select Start State" option as "current" and click on the "Update" button. In this way, we have chosen the starting position as the current stance for planning.

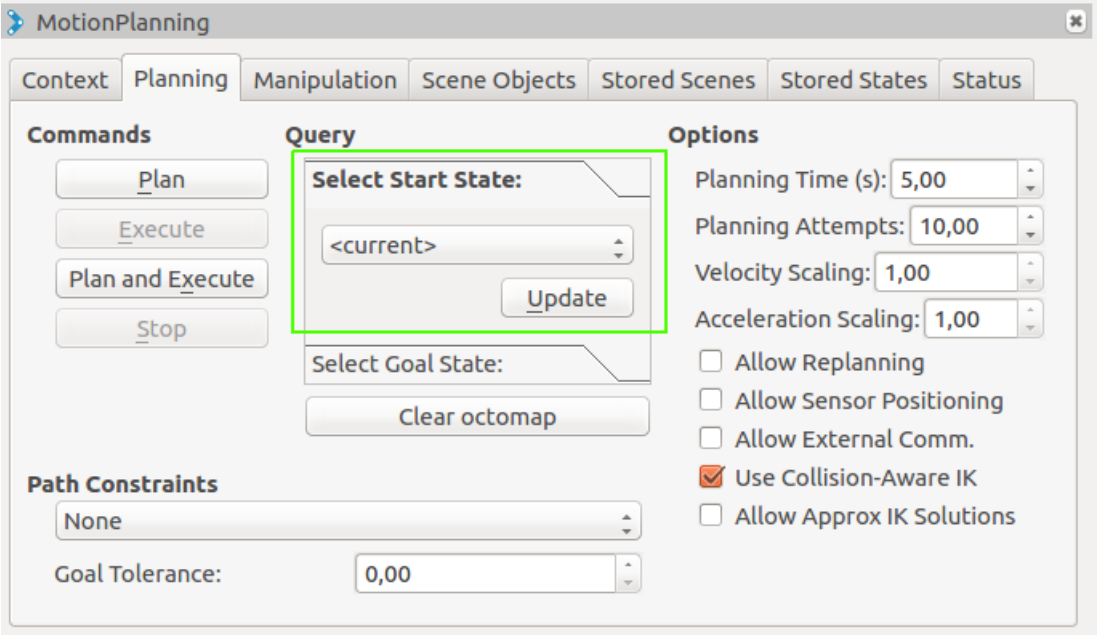

Figure 131 Start State Selection

Then, we can select the "Select Goal State" option as "random valid" and click on the "Update" button. In this way, we select the target location for planning as a random valid stance.

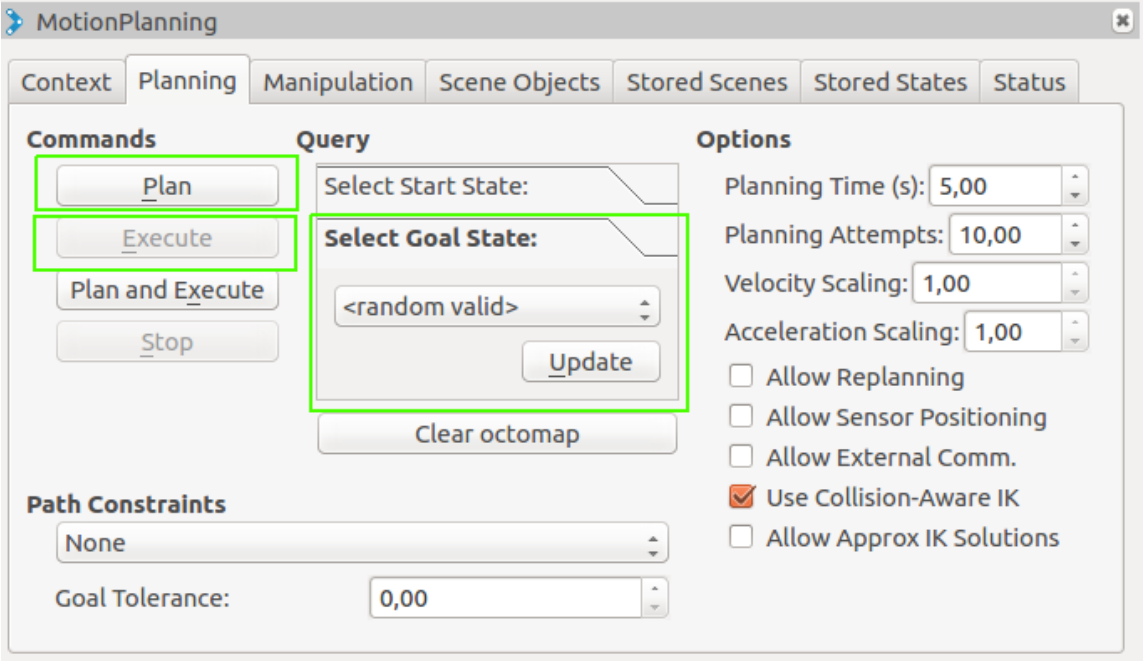

Figure 132 Random Goal State Selection

After this situation, planning is made by clicking on the "Plan" option and then the "Execute" option is implemented.

While the image formed on Rviz after the “Plan” is as follows;

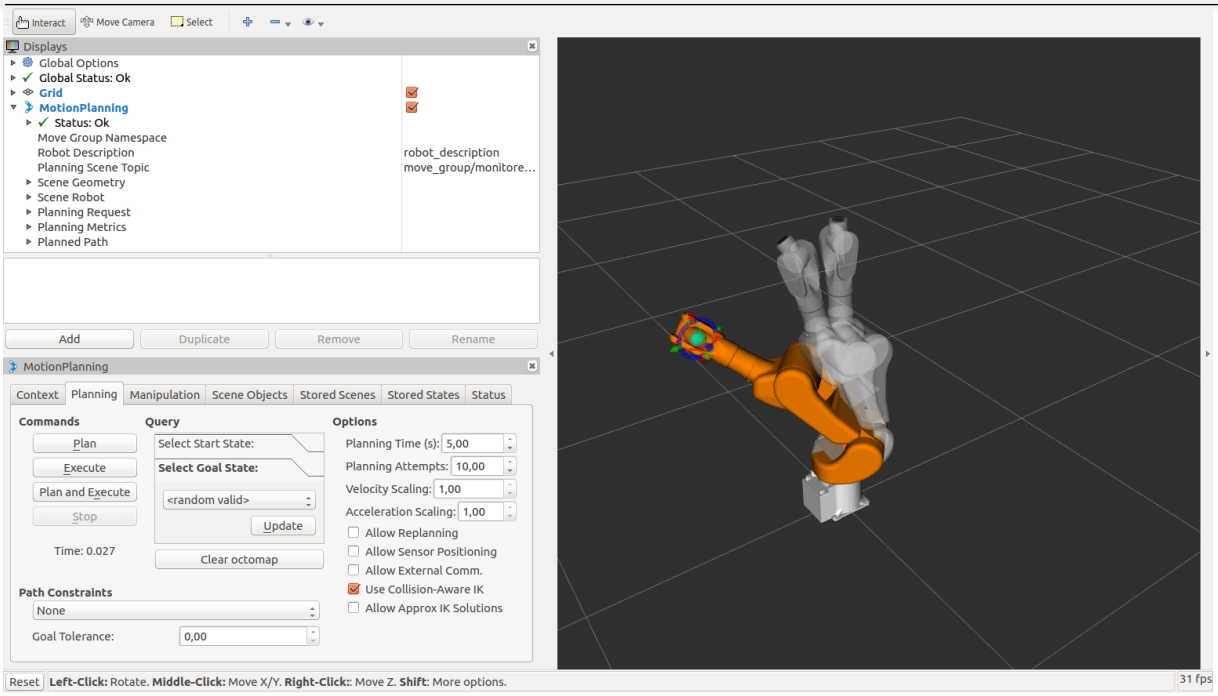

Figure 133 Created Plan

After the planned trajectory is sent to the arm controller with "Execute", it is seen that the robot reaches the target position on Gazebo;

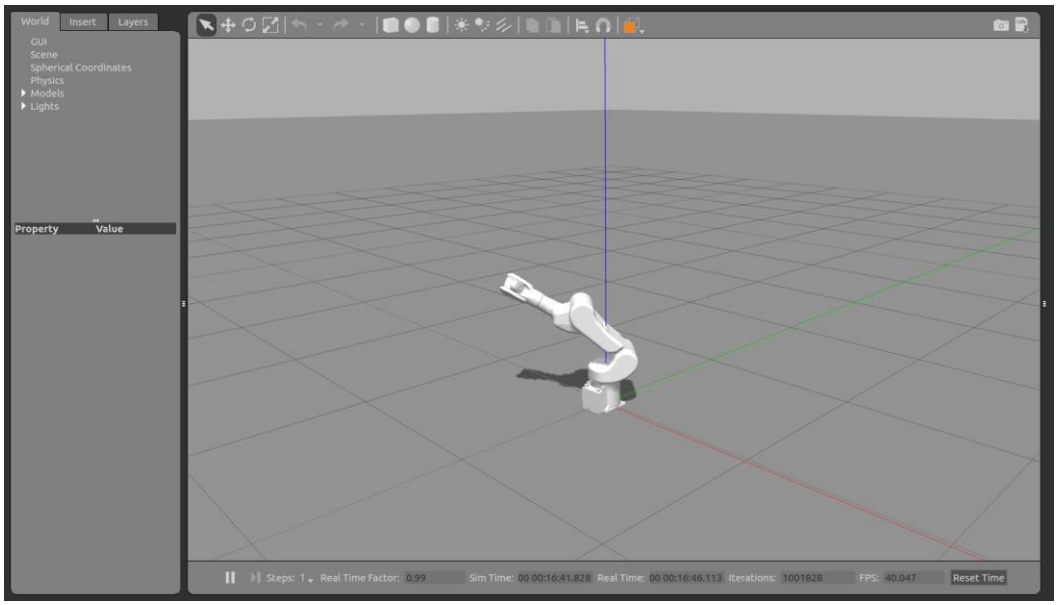

Figure 134 Visualization on Gazebo

6.3. Sample Application

For example, create a motion plan for a robot with a starting stance (green) and target stance (orange) as shown below.

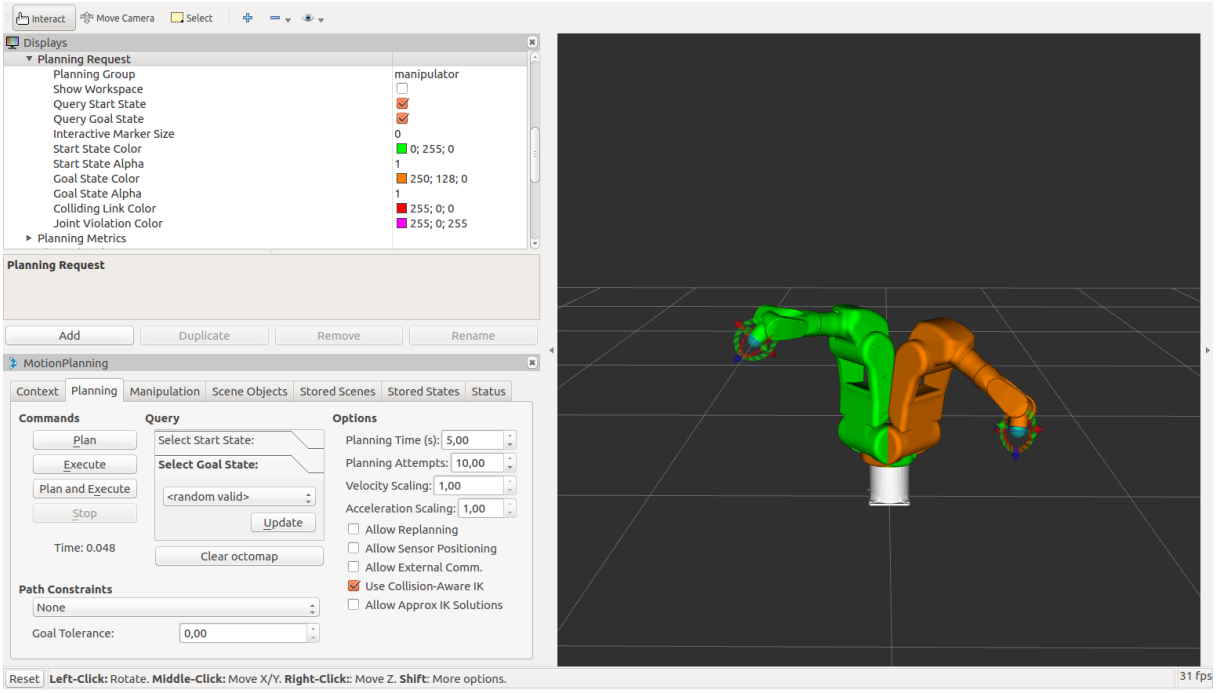

Figure 135 Sample Application

The robot stops (white) at certain moments of the orbit formed when the motion plan is created will appear on Rviz as follows. Here, “Show Trail” option is selected and “Trail Step Size” value is determined as “6”.

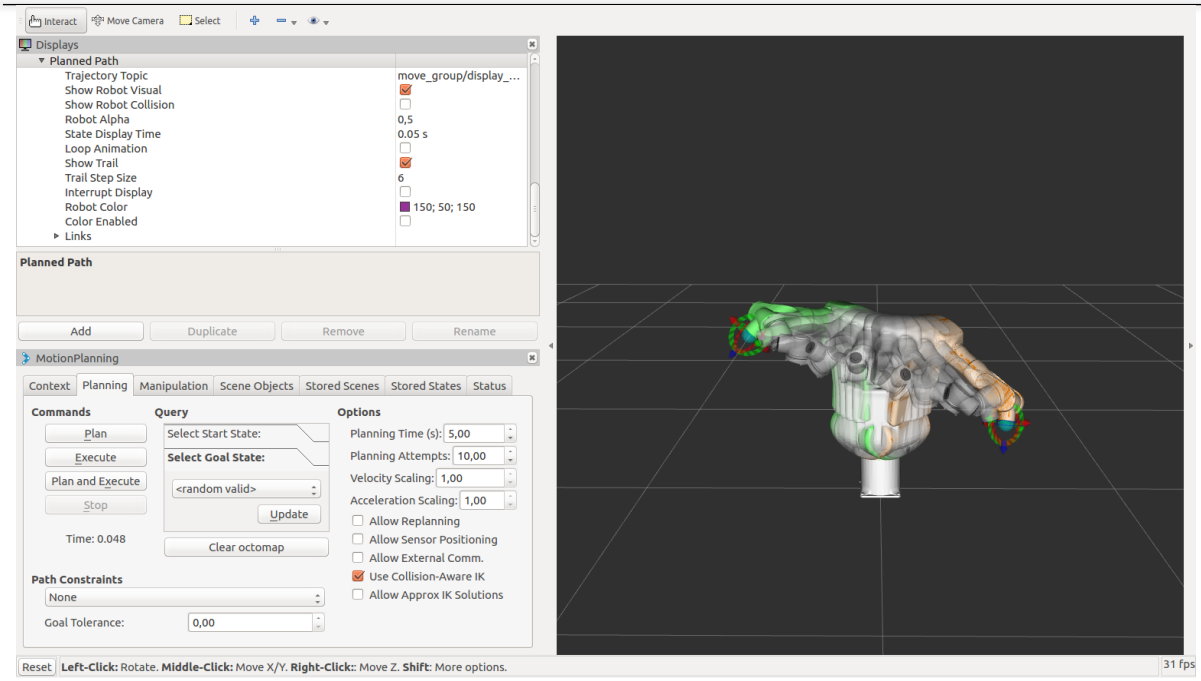

Figure 136 Sample Application

When we execute the "Execute" command on Rviz, it will be seen that the model on Gazebo goes from the starting position to the target position.

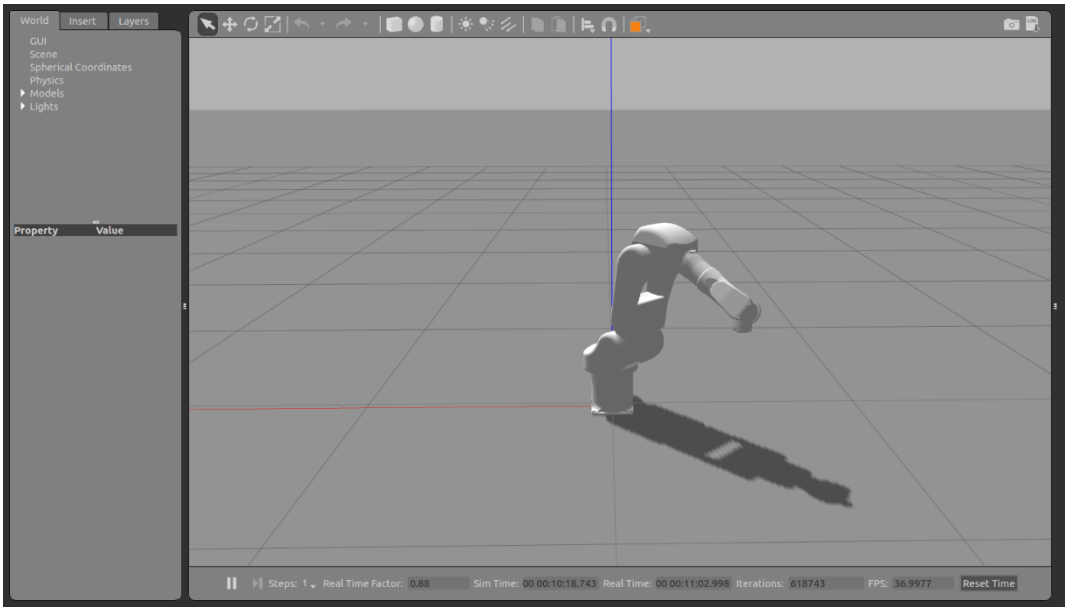

Figure 137 Sample Application