Control of IIT Kharagpur Humanoid using Robotic operating system.

Technical Report · July 2014

DOI: 10.13140/RG.2.1.4318.6645

2 authors, including:

Teja Krishna Mamidi
Healthcare Technology Innovation Centre
16 PUBLICATIONS **16** CITATIONS

SEE PROFILE

CONTROL OF IIT KHARAGPUR HUMANOID USING ROBOTIC OPERATING SYSTEM (R.O.S)

A SUMMER INTERN REPORT

Submitted by

M. Teja Krishna

Roll No: N091286

G. Vinod

Roll No: N091289

Under the guidance of

Prof. C.S.Kumar

in partial fulfillment of Summer internship for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

MECHANICAL ENGINEERING

RGUKT NUZVID Campus

Rajiv Gandhi University of Knowledge Technologies

Nuzvid, Krishna (Dist.), Andhra Pradesh

JULY 2014

Indian Institute of Technology Kharagpur
Kharagpur, Paschim Medhinpur,
West Bengal, India-721302.

CERTIFICATE

Certified that the summer internship project report "**Control of IIT Kharagpur Humanoid Using Robotic Operating System**" is the bonafide work of **M. Teja Krishna**, Roll No: N091286 and **G. Vinod**, Roll No: N091289, 3rd Year B.Tech in Mechanical Engineering of RGUKT Nuzvid Campus of Rajiv Gandhi University Knowledge Technologies (RGUKT), Andhra Pradesh Carried out under my supervision during 28.5.2014 to 23.7.2014.

Place : Kharagpur <<Signature of the Professor>>

Date **C.S. KUMAR**
PROFESSOR
MECHANICAL ENGINEERING

ACKNOWLEDGEMENTS

We are happy to express our gratitude towards our Prof. C.S. Kumar for his valuable guidance, support and to the time he spent with us. Even under his busy schedule he is with us to help us in solving our problems.

Also we would like to thank Mr. Nava Raja, Mr. Roshan and all other incharges of CAD/CAM Lab, Digital Manufacturing Lab and Robotics and Automation Lab of Mechanical Department for their encouragement and help throughout the course of the work.

We would like to thank our Lect. Satya Dev, Mechanical Engineering Department RGUKT, for helping us to communicate with Prof. C.S.Kumar and Lect. Bhasha, Mechanical Engineering Department RGUKT for helping us to adapt to the environment within the campus.

Finally, we would like to extend our gratitude towards our parents for providing mental, financial support for us in fulfilling the internship successfully without hurdles.

LIST OF FIGURES

LIST OF TABLES

LIST OF PHOTOGRAPHS

LIST OF ABBREVIATIONS

ROS: Robotic Operating System

ATDD: acceptance test-driven development

YARP: Yet Another Robot Platform

OROCOS: Open Robot Control Software

CARMEN: Carnegie Mellon Robot Navigation

MOOS: Mission Oriented Operating Suite

RPC: Remote Procedure Call

VCS: Version Control System

DNS: Domain Name Servers

URI: Uniform Resource Identifier

GUI: Graphical User Interface

XML: Extensible Markup Language

ABSTRACT

Robotics is an emerging field of science which has a wide variety of applications in many disciplines. In this field of science many robots are designed and operated using different types of frameworks. One such similar framework is ROS (Robotic Operating System). ROS is a meta operating system that supports various other frameworks. ROS supports many sensors like 2d range finders, 3d sensors, cameras and hardware like joysticks, servo motors, Lego NXT etc.

One such servo that can be handled using ROS is dynamixel servo. We can also handle these servos using some other platforms but ROS provides simple ways of communications with the servos. Humanoid at IIT KGP, India is made up of many such dynamixel servos. So using ROS is the best way to maneuver the humanoid. The present work focuses on giving the data that is generated from the human motion to the humanoid using ROS. It also concentrates on the consistency of humanoid motion.

The result of this work can be helpful in implementing various gaits on the humanoid. The autonomous level of the humanoid can also be enhanced which provides the facility to avail the humanoid in real-time applications.

TABLE OF CONTENTS

Chapter 1:

INTRODUCTION .

1.1. INTRODUCTION TO ROBOTICS:

Robotics is the branch of technology that deals with the design, construction, operation, and application of robots as well as computer systems for their control, sensory feedback, and information processing. These technologies deal with automated machines that can take the place of humans in dangerous environments or manufacturing processes, or resemble humans in appearance, behavior, and/or cognition. Many of today's robots are inspired by nature contributing to the field of bio-inspired robotics.

The word *robot* comes from the Slavic word *robota*, which means labour. Robot is a system that contains sensors, control systems, manipulators, power supplies and software all working together to perform a task. Designing, building, programming and testing robots is a combination of physics, mechanical engineering, electrical engineering, structural engineering, mathematics and computing. A study of robotics means that students are actively engaged with all of these disciplines in a deeply problem-posing problem-solving environment.

The mechanical structure of a robot must be controlled to perform tasks. The control of a robot involves three distinct phases – perception, processing, and action. Sensors give information about the environment or the robot itself (e.g. the position of its joints or its end effector). This information is then processed to be stored or transmitted, and to calculate the appropriate signals to the actuators (motors) which move the mechanical structures.

The processing phase can range in complexity. At a reactive level, it may translate raw sensor information directly into actuator commands. Sensor fusion may first be used to estimate parameters of interest (e.g. the position of the robot's gripper) from noisy sensor data. An immediate task (such as moving the gripper in a certain direction) is inferred from these estimates. Techniques from control theory convert the task into commands that drive the actuators.

1.2. ROBOT FRAMEWORKS:

Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD)[Appendix I]. Its testing capabilities can be extended by test libraries implemented either with Python or Java, and users can create new higher-level keywords from existing ones using the same syntax that is used for creating test cases. Some of these robot frameworks are Player, YARP, Orocos, CARMEN, Orca, MOOS, and Microsoft Robotics Studio. One such similar framework is ROS (Robotic operating system).

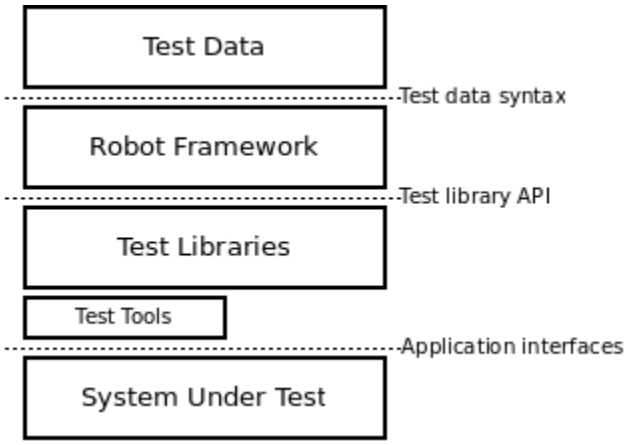

Figure: 1.1. Robot framework architecture

Architecture of a typical robot framework is as depicted in the above figure. The test data is in simple, easy-to-edit tabular format. When Robot Framework is started, it processes the test data, executes test cases and generates logs and reports. The core framework does not know anything about the target under test, and the interaction with it is handled by test libraries. Libraries can either use application interfaces directly or use lower level test tools as drivers.

Chapter 2:

LITERATURE REVIEW .

2.1. WHAT IS ROS?

ROS is an open-source, meta-operating system for robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

ROS is not a realtime framework, though it is possible to integrate ROS with realtime code. The goal of ROS is not to be a framework with the most features. Instead, the primary goal of ROS is to support code reuse in robotics research and development. ROS is a distributed framework of processes (aka Nodes) that enables executables to be individually designed and loosely coupled at runtime. These processes can be grouped into Packages and Stacks, which can be easily shared and distributed. ROS also supports a federated system of code Repositories that enable collaboration to be distributed as well. This design, from the filesystem level to the community level, enables independent decisions about development and implementation, but all can be brought together with ROS infrastructure tools.

2.1.1. *ROS Filesystem Level:*

The filesystem level concepts are

- **Packages:**

  Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (*nodes*), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most granular thing you can build and release.

- **Meta packages:**

  Meta-packages are specialized Packages which only serve to represent a group of related other packages.

- **Package manifests:**

  Manifests (package.xml) provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.

- **Repositories:**

  A collection of packages which share a common VCS system. Repositories can also contain only one package.

- **Message type:**

  Message descriptions which define the data structures for messages sent in ROS.

- **Service type:**

  Service descriptions which define the request and response data structures for services in ROS.

2.1.2. *ROS Computation Graph Level:*

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways. These concepts are implemented in the ros_comm repository.

- **Nodes:**

  Nodes are processes that perform computation. A robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

- **Master:**

  The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.

- **Parameter server:**

  The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.

- **Messages:**

  Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported.

- **Topics:**

  The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. A node sends out a message by publishing it to a given topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

- **Services:**

  Request / reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

- **Bags:**

  Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data.

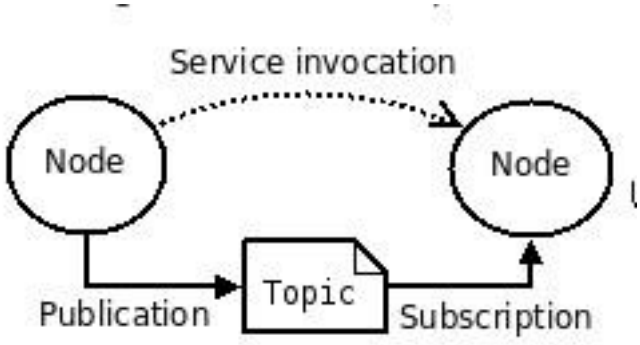

Figure: 2.1. Depiction of Computation Graph level.

Nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in ROS is called TCPROS**.**

2.1.3. *ROS Community Level:*

The ROS Community Level concepts are ROS resources that enable separate communities to exchange software and knowledge. These resources include distributions, repositories, ROS wiki etc.

2.2. COMMANDLINE TOOLS

There are various command-line tools for different operations in ROS. The commands and their uses are as follows:

- **Filesystem tools**:

  Code is spread across many ROS packages and stacks. Navigation is provided with command-line tools such as rosls and roscd in ROS. To give description about packages and stacks; rospack and rosstack command-line tools are used.

  i.rosls:

  rosls is part of the rosbash suite. It allows you to see directly the contents of a

package, stack, or common location by name rather than by package path.

Usage: $ rosls [locationname[/subdir]]

ii.roscd:

roscd is part of the rosbash suite. It allows you to change directory directly to a package or a stack.

Usage: $ roscd [locationname[/subdir]]

iii.rospack:

rospack and rosstack allow you to get information about packages and stacks.

Usage: $ rospack <command> [options] [package]

Command	Usage	Description
Depends	$ rospack depends [package]	Gives ordered list of all dependencies of the package.
depends1	$ rospack depends1 [package]	Gives first order dependencies of the package.
Find	$ rospack find [package]	Print absolute path to the package.
List	$ rospack list	Gives the list of all packages.
Profile	$ rospack profile [--length=<length>]	How many directories to display
	$ rospack profile [--zombie-only]	Only print directories that do not have any manifests.

Table. 2.1. rospack commands.

iv.rosstack:

Usage: $ rosstack <command> [options] [stack]

Command	Usage	Description
Find	$ rosstack find [stack]	Print absolute path of the stack.
List	$ rosstack list	List all stacks.
Depends	$ rosstack depends [stack]	Gives all dependencies of the stack.
contains	$ rosstack contains [package]	To check whether the package is present in the stack or not.

Table 2.2. rosstack commands.

- **Graph concept tools:**

ROS graph concepts discuss the use of roscore, rosnode, rostopic and rosrun command-line tools.

i.roscore:

roscore is a collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate. It is launched using the roscore command.

roscore will start up:

i. a ROS Master
ii. a ROS Parameter Server
iii. a rosout logging node

NOTE: If you use roslaunch, it will automatically start roscore if it detects that it is not already running.

Usage: $ roscore

ii.rosnode:

rosnode is a command-line tool for displaying debug information about ROS Nodes, including publications, subscriptions and connections. It also contains an experimental library for retrieving node information.

Usage: $ rosnode <command> [options] <node>

Command	Usage	Description
Info	$ rosnode info <node-name>	Display information about a node, including publications and subscriptions.
Kill	$ rosnode kill <node-name>	Kill one or more nodes by name.
List	$ rosnode list	Display a list of current nodes.
machine	$ rosnode machine <machine-name>	List nodes running on a particular machine.
Ping	$ rosnode ping <node-name>	Test connectivity to a node.
cleanup	$ rosnode cleanup	Purge registration information of unreachable nodes

Table 2.3. rosnode commands

iii. rosrun:

rosrun allows you to run an executable in an arbitrary package from anywhere without having to give its full path or cd/roscd there first.

Usage: $ rosrun <package> <executable>

iv.rostopic:

The rostopic command-line tool displays information about ROS topics. Currently, it can display a list of active topics, the publishers and subscribers of a specific topic, the publishing rate of a topic, the bandwidth of a topic, and messages published to a topic.

Usage: $ rostopic <command> [options] <topic-name>

v.rosservice:

The rosservice command implements a variety of commands that let you discover which services are currently online from which nodes and further drill down to get specific information about a service, such as its type, URI, and arguments.

Usage: $ rossevice <command> [options] <service-name>

vi.rosmsg:

rosmsg is a command-line tool for displaying information about ROS Message types.

Usage: $ rosmsg <command> [options]

vii.rossrv:

rossrv is a command-line tool for displaying information about ROS Service types.

Usage: $ rossrv <command> [options]

2.3. CREATING A SIMPLE ROS PACKAGE

All ROS packages consist of the many similar files: manifests, CMakeLists.txt, mainpage.dox, and Makefiles. roscreate-pkg command tool eliminates many tedious tasks of creating a new package by hand, and eliminates common errors caused by hand-typing build files and manifests.

To create a new package in the current directory:

Usage: # roscreate-pkg [package_name] [depend1] [depend2] [depend3]

depend1, depend2 and depend3 are dependencies of the package.

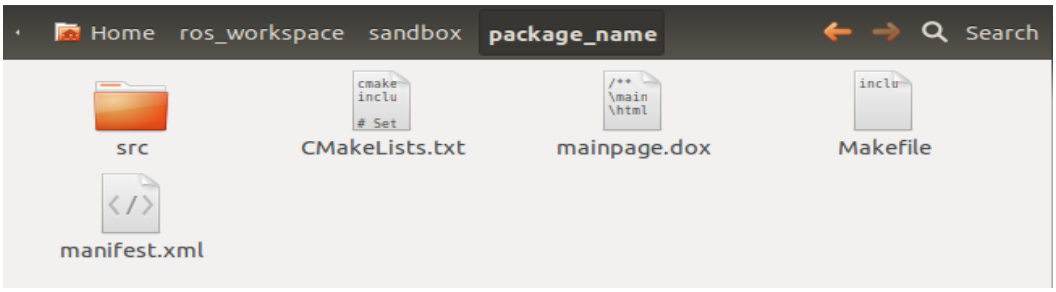

Figure 2.2. Contents of a package after creating

rosmake:

rosmake is a ros dependency aware build tool which is used to build all dependencies in the correct order.

Usage: # rosmake [options] [package] ……

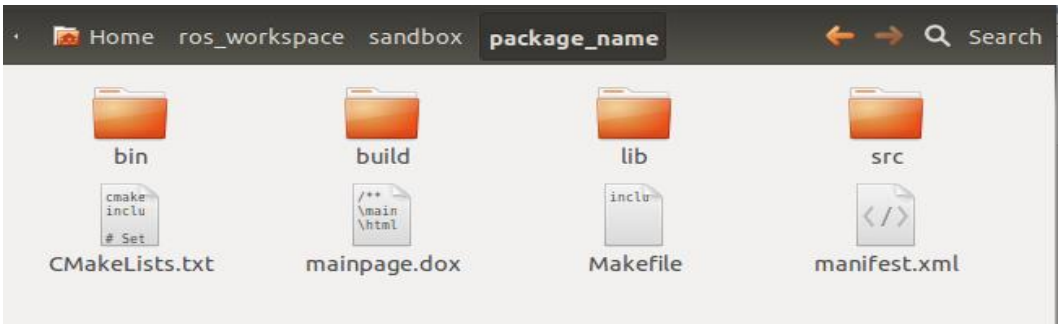

Figure 2.3. Contents of a package after building

2.4. TRACING ERRORS

For debugging purpose we can use rqt package. In this package there are different tools like rqt_console, rqt_logger_level, rqt_graph etc.

i.rqt_console:

rqt_console is a viewer in the rqt package that displays messages being published to rosout. It collects messages over time, and lets you view them in more detail, as well as allowing you to filter messages by various means.

Usage: $ rqt_console or $ rosrun rqt_console rqt_console

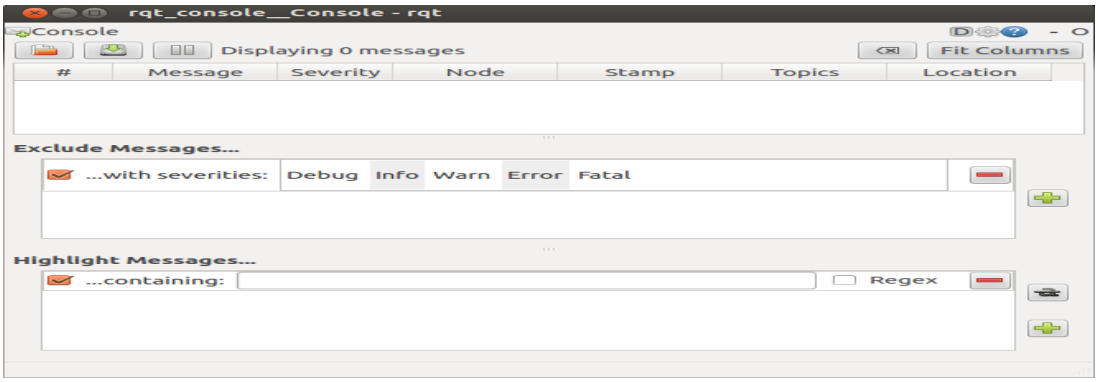

Figure 2.4. GUI window of rqt_console.

ii.rqt_logger_level:

rqt_logger_level is an application for adjusting the logger level of ros nodes. rqt_console attaches to ROS's logging framework to display output from nodes. rqt_logger_level allows us to change the verbosity level (DEBUG, WARN, INFO, and ERROR) of nodes as they run.

Usage: $ rqt_logger_level or $ rosrun rqt_logger_level rqt_loggerlevel

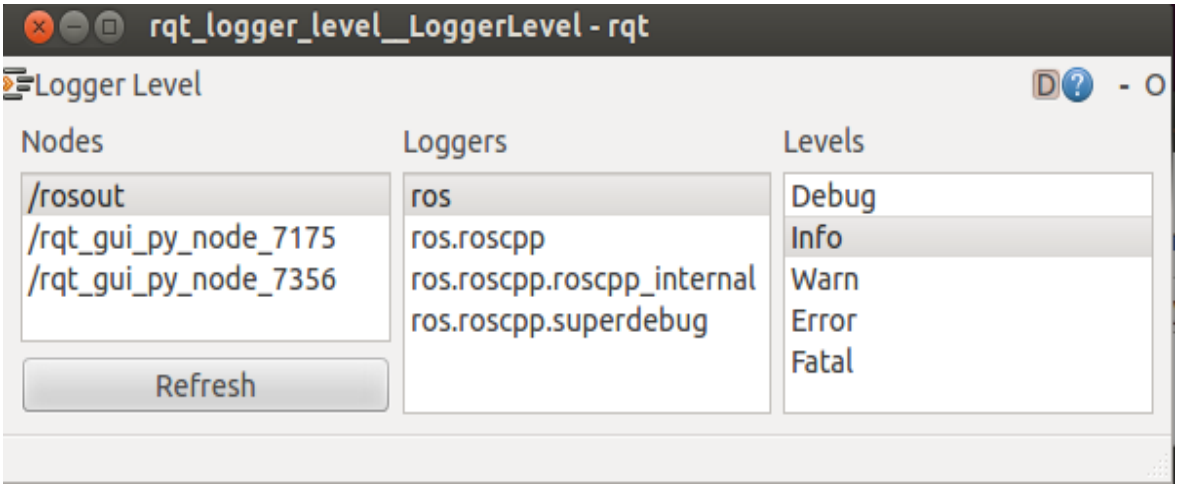

Figure 2.5. GUI window of rqt_logger_level.

iii.rqt_graph

rqt_graph provides a GUI plugin for visualizing the ROS computation graph.

Usage: $ rqt_graph or $ rosrun rqt_graph rqt_graph

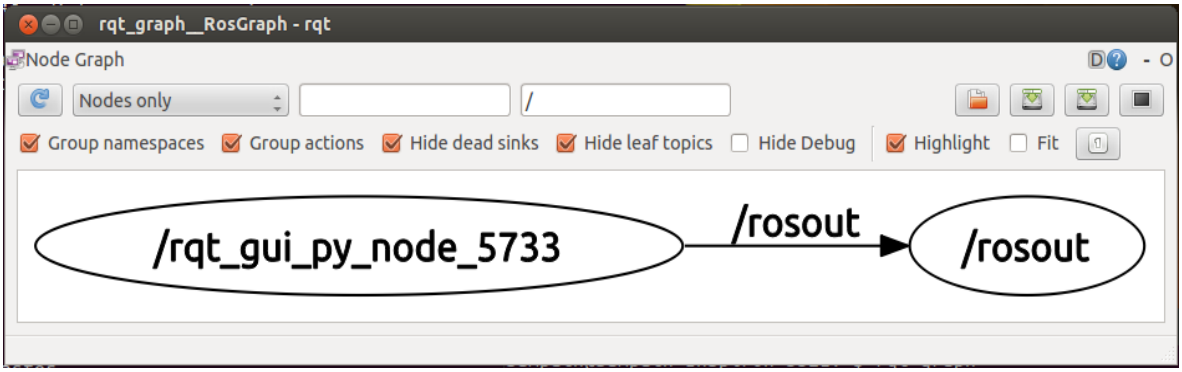

Figure 2.6. GUI window of rqt_graph.

2.5. MULTIPLE NODES

We can initialize multiple nodes at a time using launch file without executing master and nodes in individual windows. This process can be done using the command roslaunch.

roslaunch:

roslaunch takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.

Usage: $ roslaunch [package] [file_name.launch]

Tags:

In a launch file we will find many tags. Some of these tags and their usage are as listed below.

i. The <**launch**> tag is the root element of any roslaunch file. It's sole purpose is to act as a container for the other elements.
ii. The <**node**> tag is used to launch a node.
iii. The <**param**> tag sets a parameter on the parameter server.
iv. The <**remap**> tag declares a name remapping.
v. The <**machine**> tag declares a machine to use for launching.
vi. The <**rosparam**> tag sets ROS parameter for the launch using a rosparam file.
vii. The <**include**> tag includes other ROS launch files.
viii. The <**env**> tag specifies an environment variable for launched nodes.
ix. The <**test**> tag is used to launch a test node.
x. The <**arg**> tag declares an argument.
xi. The <**group**> tag is used to group enclosed elements sharing a namespace/remap.

2.6. RECORDING AND PLAYING BACK DATA

There is a provision to record data from a running ROS system into a .bag file, and then to play back the data to produce similar behavior in a running system.

rosbag:

The rosbag package provides a command-line tool for working with bags as well as code APIs for reading/writing bags in C++ and Python. rosbag is the tool to record from and play back to ROS topics.

Usage: $ rosbag <command> [options] [topic_name]

Command	Usage	Description
Record	$ rosbag record [options] <topic-names>	Record a bag file with the contents of the specified topics.
Info	$ rosbag info [options] <bag-files>	Display a summary of the contents of the bag files.
Play	$ rosbag play [options] <bag-files>	Play back (publish) the contents of the given bags.
Check	$ rosbag check [options] <bag-files>	Determine whether or not a bag is playable in the current system.

Table 2.4. rosbag commands

Chapter 3:

IMROVEMENT TO THE EXISTING SYSTEM .

3.1. UNDERSTANDING A PUBLISHER AND SUBSCRIBER

- **Publisher:**

It is a node which register its identity with the master and sends data on a topic. Here is a node which publishes a string "hello world" on a topic 'chatter'.

publisher.py

```
1  /usr/bin/env python
2  import rospy
3  from std_msgs.msg import String
4
5  def talker():
6      pub = rospy.Publisher('chatter', String, queue_size=10)
7      rospy.init_node('talker')
8      r = rospy.Rate(10) # 10hz
9      while not rospy.is_shutdown():
10         str = "hello world %s"%rospy.get_time()
11         rospy.loginfo(str)
12         pub.publish(str)
13         r.sleep()
14
15 if __name__ == '__main__':
16     try:
17     talker()
18 except rospy.ROSInterruptException: pass
19
```

Figure 3.1. A publisher node.

i. The first line makes sure your script is executed as a Python script.

ii. We need to import rospy to write a ROS node in python because rospy provides many modules for publishing. We also need to import data structure string from the msg folder in std_msgs package.

iii. The line six declares that the node is publishing to the chatter topic using the message type String. The queue_size argument has been added in *Hydro* and limits the amount of queued messages if any subscriber is not receiving them fast enough.

iv. The next line, rospy.init_node(NAME), is very important as it tells rospy the name of the node and until rospy has this information, it cannot start communicating with the ROS Master.

v. The line eight creates a Rate object r. With the help of its method sleep(), it offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second.

vi. The while loop is a fairly standard rospy construct: checking the rospy.is_shutdown() flag and then doing work. The "work" is a call to pub.publish(String(str)) that publishes to our chatter topic using a newly created String message. The loop calls r.sleep(), which sleeps just long enough to maintain the desired rate through the loop. This loop also calls rospy.loginfo(str), which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout.

- **Subscriber:**

Subscriber is a node which looks for an appropriate identity of a publisher at the master to receive data from an appropriate topic.

Here is a node which subscribes to a topic named 'chatter'

subscriber.py

```
1   #!/usr/bin/env python
2   import rospy
3   from std_msgs.msg import String
4
5   def callback(data):
6       rospy.loginfo(rospy.get_caller_id()+"I heard %s",data.data)
7
8   def listener():
9       rospy.init_node('listener', anonymous=True)
10      rospy.Subscriber("chatter", String, callback)
11      rospy.spin()
12
13  if __name__ == '__main__':
14      listener()
15
```

Figure 3.2. A subscriber node.

i. In line nine, rospy.init_node(NAME), is very important as it tells rospy the name of the node until rospy has this information, it cannot start communicating with the ROS

Master. The anonymous=True flag tells rospy to generate a unique name for the node so that you can have multiple listener.py nodes run easily.

ii. The next line tells that the node subscribes to a topic name 'chatter' with message type string and sends the data received to a callback object.

iii. In lines five and six the callback function prints the received data through rospy.loginfo.

iv. rospy.spin() simply keeps your node from exiting until the node has been shutdown.

3.2. A RANDOM INTEGER PUBLISHER AND SUBSCRIBER

In the previous section we discussed about a publisher that will generate a string "hello world" on a topic and a subscriber that receives the string on the same topic.

Here we will see a node which can publish a random integer and a node which can subscribe to that random integer.

- **Random integer publisher:**

Random_integer_publisher.py

```
1  #!/usr/bin/env python
2
3  import rospy
4  from std_msgs.msg import Float64
5  import random
6
7  def talker():
8      rospy.init_node("Random_integer_publisher")
9      pub = rospy.Publisher('random', Float64)
10     r =rospy.Rate(10)
11     while not rospy.is_shutdown():
12         int = random.randint(1,100)
13         pub.publish(int)
14         print int
15         r.sleep()
16
17 if __name__ == '__main__':
18     try:
19         talker()
20     except rospy.ROSInterruptException: pass
21
```

Figure 3.3. A Random integer publisher node.

i. Here we import a module named random which provides object that can generate a random integer.

ii. In the while loop randint object from random module is used to generate random integer between 1 and 100. This random integer is published to the topic random.

- **Random integer subscriber:**

i. Here we initialized a node named as Random_integer_subscrier.

ii. This node subscribes to a random integer through the topic named random.

Random_integer_subscriber.py

```
1  #! /usr/bin/env python
2
3  import rospy
4  from std_msgs.msg import Float64
5
6  def callback(data):
7      print data.data
8
9  if __name__ == '__main__':
10     rospy.init_node("Random_integer_subscriber")
11     rospy.Subscriber("random", Float64, callback)
12     rospy.spin()
13
14
```

Figure 3.4. A Random integer subscriber node.

When the above two nodes are executed the publisher publishes the topic and subscriber receives that topic. The computation graph which explains the process is as shown below

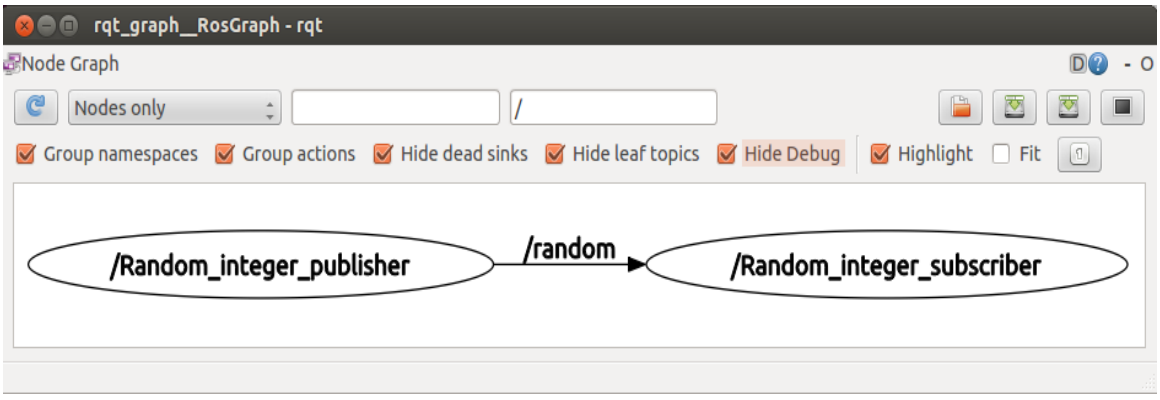

Figure 3.5. rqt_graph of random integer publisher and subscriber.

3.3. A SINGLE NODE TO PUBLISH AND SUBSCRIBE

In the previous sections we saw nodes which can either publish or subscribe. Now in this section we will see a node which subscribes the data and publishes that subscribed data.

Here this node subscribes a random integer from a topic named ‘topic1’ and publishes this random integer on a new topic named ‘topic2’

pub_sub.py

```
1   #! /usr/bin/env python
2
3   import rospy
4   from std_msgs.msg import Float64
5
6   def callback(data):
7       print data.data
8       pub = rospy.Publisher("topic2", Float64)
9       pub.publish(data.data)
10
11  if __name__ == '__main__':
12      rospy.init_node("pub_sub")
13      rospy.Subscriber("topic1", Float64, callback)
14      rospy.spin()
```

Figure 3.6. A single node to publish and subscribe.

i. In the callback function we are intimating the master that this node along with subscribing it also publishes a float value on topic2 using rospy.Publisher(“topic2”, Float64) and we are publishing the subscribed data using pub.publish(data.data).

- **rqt_graph for pub_sub:**

  When the above node along with the two other nodes, one for publishing the random integer on ‘topic1’ and the other for receiving the random integer from pub_sub node on ‘topic2’ are executed, the computation graph is as shown below

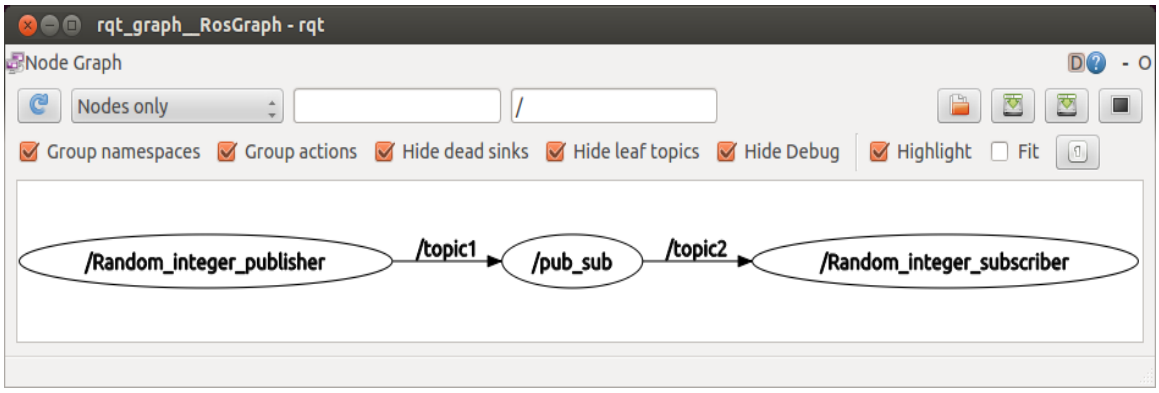

Figure 3.7. rqt_graph of pub_sub node.

3.4. USAGE OF LAUNCH FILE

Instead of running roscore, publisher node and subscriber node in different shells we can run all these executables in a single shell using launch file and that launch file is as shown.

file_name.launch

```
<launch>
    <node name="publisher_name" pkg="package_name" type="publisher_executable_file_name" output="screen"/>
    <node name="subscriber_name" pkg="package_name" type="subscriber_executable_file_name" output="screen"/>
</launch>
```

Figure 3.8. A general launch file to run publisher and subscriber.

Example1:

random.launch

```
<launch>
    <node name="Random_integer_publisher" pkg="test" type="talker.py" output="screen"/>
    <node name="Random_integer_subscriber" pkg="test" type="listener.py" output="screen"/>
</launch>
```

Figure 3.9. A launch file to run a random integer publisher and subscriber.

Here the node names are Random_intger_publisher and Random_integer_subscriber. These nodes are in a package named as test with file names talker.py and listener.py respectively.

Example2:

random.launch

```
<launch>
    <node name="Random_integer_publisher" pkg="test" type="talker.py" output="screen"/>
    <node name="Random_integer_subscriber1" pkg="test" type="listener.py" output="screen"/>
    <node name="Random_integer_subscriber2" pkg="test" type="listener.py" output="screen"/>
    <node name="Random_integer_subscriber3" pkg="test" type="listener.py" output="screen"/>
</launch>
```

Figure 3.10. A launch file to run a random integer publisher and many subscribers.

The node named Random_integer_publisher is a publisher. From this publisher three nodes subscribe the data on a topic that is published by the publisher. These three different nodes are generated from a single executable file with the help of launch file.

The computation graph for this launch file is generated using the command rqt_graph in a new shell and the graph is as shown below,

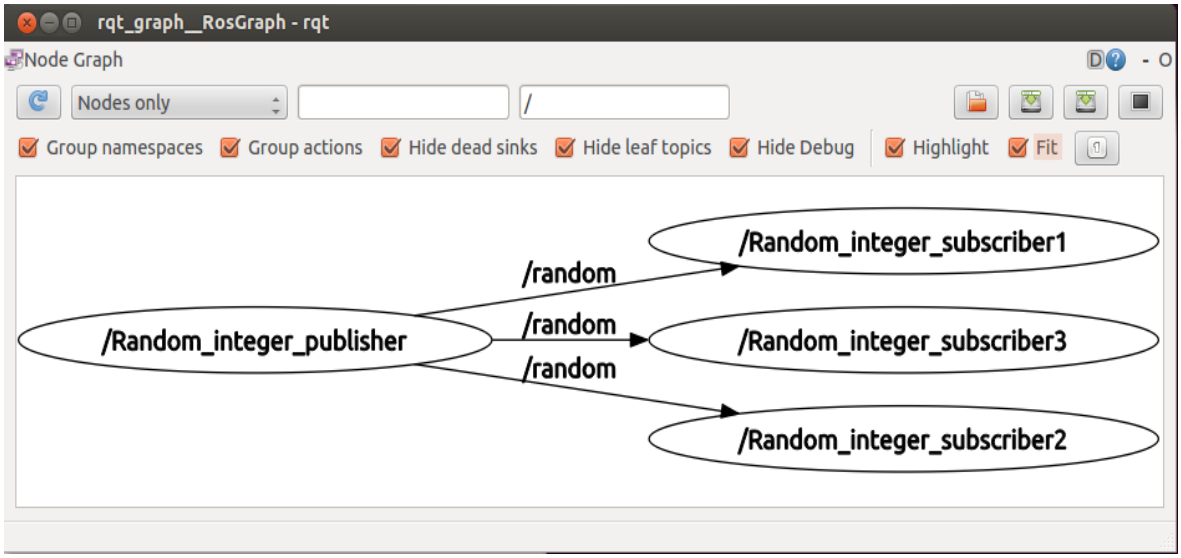

Figure 3.11. rqt_graph for one publisher and three subscribers.

Like this we can send the data on a single topic from one publisher to many subscribers. To generate different subscribers there is no need to write different programs. We can generate these from a single executable node.

Example3:

lis_tal.launch

```
<launch>
<group ns="part1">
     <node name="talker" pkg="test" type="talker.py"/>
     <node name="listener1" pkg="test" type="listener.py" output="screen"/>
</group>
<group ns="part2">
      <node name="listener2" pkg="test" type="listener.py" output="screen">
      <remap from="chatter" to="/part1/matter"/>
      </node>
</group>
</launch>
```

Figure 3.12. A launch file to remap the topics.

In the above example we introduce the tag <remap>. The <remap> tag allows you to pass in name remapping arguments to the ROS node that you are launching in a more structured manner than setting the args attribute of a <node> directly. The <remap> tag applies to all subsequent declarations in its scope. Its usage is,

from = “original name” (name that you are remapping) to = “ new name” (target name).

The computation graph of this launch file is as shown below. In the below rqt_graph /part1/talker publishes message on a topic /part1/chatter and /part1/listener subscribes to the topic /part1/chatter and publishes to /part1/matter. /part2/listener2 subscribes to the topic /part2/chatter.

Since /part2/chatter and /part1/matter are of same type we remapped it using remap tag.

Figure 3.13. rqt_graph which illustrates the remap tag.

Chapter4:

CONTROL IIT KHARAGPUR HUMANOID

4.1. IIT KHARAGPUR HUMANOID

A **humanoid** is something that has an appearance resembling a human being. A humanoid design might be for functional purposes, such as interacting with human tools and environments, for experimental purposes, such as the study of bipedal locomotion, or for other purposes. In general, humanoid robots have a torso, a head, two arms, and two legs, though some forms of humanoid robots may model only part of the body, for example, from the waist up. Some humanoid robots may also have heads designed to replicate human facial features such as eyes and mouths.

The IIT Kharagpur Humanoid is the first adult size humanoid robot in South Asia. It has a height of 115cm (including the head) and has 22 degrees of freedom, with 6 degrees per leg and the rest in the upper body. The robot uses very powerful EX-106, RX-64 and AX-12A intelligent actuators with Maxon motors and internal Motion control board enabling voltage, current, position and torque feedback; with each motor having a working torque of the range of 64 to 106 kg.cm.

The body of the humanoid is made out of precision laser-cut parts of 3000 series Aluminium Manganese alloy, providing strength and high workability. Smaller parts are made by Subtractive rapid prototyping of HDPE, thereby decreasing the time between design to a working prototype from days to hours. The humanoid has advanced sensing capabilities including a torso mounted MEMS Gyro (6 DOF sensor with 9 states) and an array of pressure sensors on each foot to allow for precise detection and adjustment of the Centre of Pressure there by enabling the fall-detection and avoidance algorithm to work.

Photo 4.1. IIT Kharagpur humanoid.

The IIT Kharapur humanoid joints are actuated by dynamixel motors like AX-12A, RX-64 and EX-106.

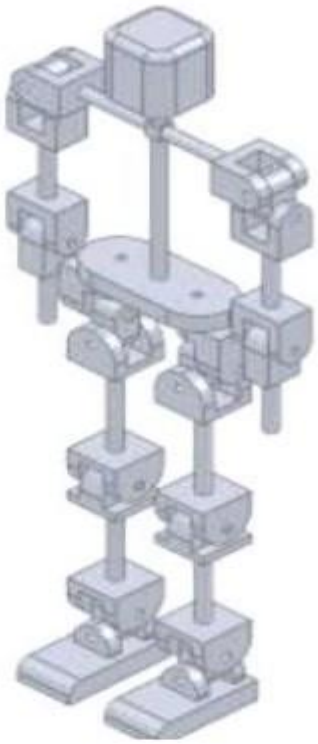

Figure 4.1. Actuator positions in IIT Kharagpur humanoid.

4.2. INTRODUCTION TO DYNAMIXEL SERVOS

DYNAMIXEL (DXL) is a line-up high performance networked actuators for robots developed by a Korean manufacturer ROBOTIS. DXL is being used by numerous companies, universities, and hobbyist due to its versatile expansion capability, powerful feedback functions, position, speed, internal temperature, input voltage, etc. and it's daisy chain topology for simplified wiring connections.

DXL can be used for multi-joint robot systems such as robotic arms, robotic hand, bi-pedal robot, hexapod robot, snake robot, scorpions, pan tilts, kinematic art, animatronics and automation, etc... In most servos a gearbox is attached to increase the output torque. This output then has generally a position sensor to measure its position for feedback to the controller.

There are various types of dynamixel servos. These servos are shown in below figure.

Photo 4.2. Types of dynamixel servos.

The specifications of these dynamixel servos are

Dynamixel	Stall Torque (Nm)	Position Sensor (Resolution)	Network I/F	Motor	Gear Ratio (Material)
AX-12W	N/A	Potentiometer 300/1024	TTL	Cored Motor	1:32(empla)
AX-12A	1.5 at 12V	Potentiometer 300/1024	TTL	Cored Motor	1:254(empla)
AX-18A	1.8 at 12V	Potentiometer 300/1024	TTL	Coreless Motor	1:254(empla)
MX-28	2.5 at 12V	Contactless Absolute Encoder 360/4096	TTL/RS485	Maxon Motor	1:193(metal)
RX-24F	2.6 at 12V	Potentiometer 300/1024	RS485	Coreless Motor	1:193(metal)
RX-28	3.7 at 18.5V	Potentiometer 300/1024	RS485	Maxon Motor	1:193(metal)
MX-64	6 at 12V	Contactless Absolute Encoder 360/4096	TTL/RS485	Maxon Motor	1:200(metal)
RX-64	5.3 at 18.5V	Potentiometer 300/1024	RS485	Maxon Motor	1:200(metal)
MX-106	8.4 at 12V	Contactless Absolute Encoder 360/4096	TTL/RS485	Maxon Motor	1:225(metal)
EX-106+	10.9 at 18.5V	Magnetic Encoder 251/4096	RS485	Maxon Motor	1:184(metal)

Table 4.1. Specifications of dynamixel servos.

IIT Kharagpur humanoid arms are actuated by AX-12A servos. It can produce high torque and is

made with high quality materials to provide the necessary strength and structural resilience to withstand large external forces. It also has the ability to detect and act upon internal conditions such as changes in internal temperature or supply voltage. AX-12A is a new version of the AX-12+ with the same performance but more advanced external design.

Photo 4.3. AX-12A dynamixel servo.

The Dynamixel servo gets its set-points via a serial communication bus. These set-points are then processed to drive the electric motor. To establish a serial communication from PC to dynamixel a usb to serial convertor is used. USB2Dynamixel is a device to operate Dynamixel directly from PC. USB2Dynamixel is connected to USB port of PC, 3P and 4P connectors are installed so that various Dynamixels can be connected.

Photo 4.4.USB to serial convertor and daisy chain connection.

- **Communication:**

  The main controller sends an instruction packet with the ID set to N, only the Dynamixel unit with this ID value will return its respective status packet and perform the required instruction.

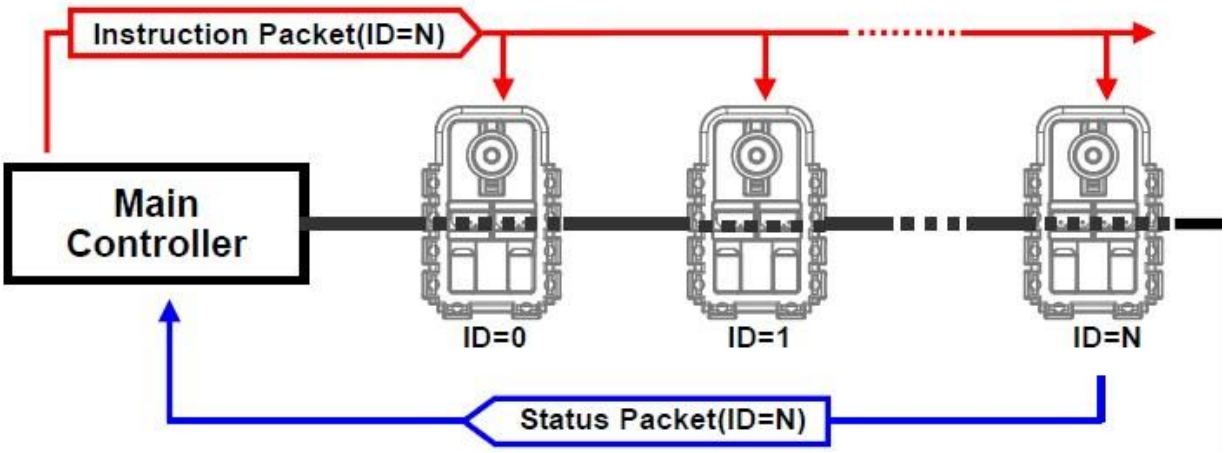

Figure 4.2. Communication between controller and dynamixel servos.

4.4. PROBLEM STATEMENT

IIT Kharagpur humanoid robot arms are actuated by dynamixel AX-12A servos. Our project is to establish a communication between ROS and these dynamixel AX-12A servos. Further, to extend this communication to the leg of the humanoid that has dynamixel RX-64, EX-106 servos.

- Task 1:

  Publish data to AX-12A servo and subscribe its state.

- Task 2:

  Publish data to one AX-12A servo by taking feedback from the other AX-12A.

- Task 3:

  Establish a mimic operation between two arms of the humanoid.

- Task 4:

  Operating humanoid arms by taking input through excel (.xls) sheet.

Chapter5:

METHODOLOGY ADOPTED

5.1. DYNAMIXEL PACKAGE

Dynamixel packages are available in a dynamixel_motor stack of ROS. This stack contains packages that are used to interface with Robotis Dynamixel line of servo motors. This stack was tested with and fully supports AX-12, AX-18, RX-24, RX-28, MX-28, RX-64, MX-64, EX-106 and MX-106 models.

To install dynamixel_motor stack on **ROS Hydro** execute the command below:

```
sudo apt-get install ros-hydro-dynamixel-motor
```

The dynamixel_motor stack contains the following packages.

- dynamixel_controllers
- dynamixel_driver
- dynamixel_msgs
- dynamixel_tutorials

➢ **dynamixel_controllers:**

This package contains a configurable node, services and a spawner script to start, stop and restart one or more controller plugins. Reusable controller types are defined for common Dynamixel motor joints. Both speed and torque can be set for each joint. This python package can be used by more specific robot controllers and all configurable parameters can be loaded via a yaml file.

➢ **dynamixel_driver:**

This package provides low level IO for Robotis Dynamixel servos. Fully supports and was tested with AX-12, AX-18, RX-24, RX-28, MX-28, RX-64, EX-106 models. Hardware specific constants are defined for reading and writing information from/to Dynamixel servos. This low level package won't be used directly by most ROS users. The

higher level dynamixel_controllers and specific robot joint controllers make use of this package.

This directory contains scripts that are useful when working with Dynamixel servos. Few among them are,

- set_torque.py -- use to turn motor torque on/off
- change_id.py -- use to change the id of a single motor
- set_servo_config.py -- use to change various servo parameters, like baud rate, return delay time, angle limits, etc.
- info_dump.py -- use to print out information for connected motors

- **dynamixel_msgs:**

This package contains common messages used throughout dynamixel_motor stack.

- **dynamixel_tutorials:**

To start with dynamixel servos we use this package. We can write our own nodes in this package without editing other package contents.

5.2. OPERATE DYNAMIXEL SERVOS

5.2.1. Giving Input To Single Servo:

In this section we will discuss the way to give input to the dynamixel servo. First run the controller_manager launch file which is in the dynamixel_tutorials package in a new shell using the below command.

```
$ roslaunch dynamixel_tutorials controller_manager.launch
```

The launch file is shown in the below code

controller_manager.launch

```
<!-- -*- mode: XML -*- -->

<launch>
    <node name="dynamixel_manager" pkg="dynamixel_controllers" type="controller_manager.py" required="true" output="screen">
        <rosparam>
            namespace: dxl_manager
            serial_ports:
                pan_tilt_port:
                    port_name: "/dev/ttyUSB0"
                    baud_rate: 1000000
                    min_motor_id: 1
                    max_motor_id: 25
                    update_rate: 20
        </rosparam>
    </node>
</launch>
```

Figure 5.1.controller_manager launch file.

After running above launch file in a shell we will see the motors detected by the manager. In present case a motor with ID 1 is detected and configured. Controller_manager node is useful to ping the motors between a range of ID's and to find the motors having individual ID's.

```
process[dynamixel_manager-2]: started with pid [2934]
[INFO] [WallTime: 1405321240.534991] pan_tilt_port: Pinging motor IDs 1 through
25...
[INFO] [WallTime: 1405321242.572589] pan_tilt_port: Found 1 motors - 1 AX-12 [1]
, initialization complete.
```

Figure 5.2. Output of controller_manager launch file.

In the above figure we can see that controller_manager node pings the motors between the range 1 and 25 and finds the motor having ID 1. However, we cannot move the motors with the topics generated by this launch file, so we need to run the next launch file to do it.

Now in a new shell run the controller_spawner launch file. Controller_spawner node is useful to establish communication for different motors. This launch file will create the necessary topics to move the motors.

The controller_spawner launch file is as shown below

controller_spawner.launch

```
<!-- -*- mode: XML -*- -->

<launch>
    <!-- Load controller configuration to parameter server -->
    <rosparam file="$(find dynamixel_tutorials)/config/dynamixel_joint_controllers.yaml" command="load"/>

    <!-- start specified joint controllers -->
    <node name="dynamixel_controller_spawner" pkg="dynamixel_controllers" type="controller_spawner.py"
          args="--manager=dxl_manager
                --port=pan_tilt_port
                --type=simple
                motor1"
          output="screen"/>
</launch>
```

Figure 5.3. controller_spawner launch file.

The loaded .yaml file is as shown below

dynamixel_joint_controllers.yaml

```
motor1:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: tilt_joint
    joint_speed: 0.5
    torque_limit: 1
    torque: 1023
    motor:
        id: 1
        init: 512
        min: 0
        max: 1023
```

Figure 5.4. dynamixel_joint_controllers.yaml file.

In this yaml file we can set parameters for shaft speed, torque limit value, maximum torque and motor specifications.

The command to run controller_spawner launch file is as follows

```
$ roslaunch dynamixel_tutorials controller_spawner.launch
```

If the process has gone well without any errors the output should be like this

```
process[dynamixel_controller_spawner-1]: started with pid [3470]
[INFO] [WallTime: 1405321698.629032] pan_tilt_port controller_spawner: waiting f
or controller_manager dxl_manager to startup in global namespace...
[INFO] [WallTime: 1405321698.634531] pan_tilt_port controller_spawner: All servi
ces are up, spawning controllers...
[INFO] [WallTime: 1405321698.681873] Controller motor1 successfully started.
[dynamixel_controller_spawner-1] process has finished cleanly
log file: /home/iitkgp/.ros/log/90161840-0b24-11e4-b435-0019d1a94093/dynamixel_c
ontroller_spawner-1*.log
all processes on machine have died, roslaunch will exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

Figure 5.5. Output of controller_spawner launch file.

After running the above two launch files the motors are identified and the necessary topics are generated to move the motors. We can move a single motor using the command rostopic pub command.

The command is

$ rostopic pub /motor1/command std_msgs/Float64 -- 0.5

The command publishes 0.5 radians of motor position with the message type std_msgs/Float64 on the topic /motor1/command.

Once the command is executed, you will see the motor moving and it will stop at 0.5 radians or 28.6478898 degrees.

The computation graph for the above published topic is as shown below

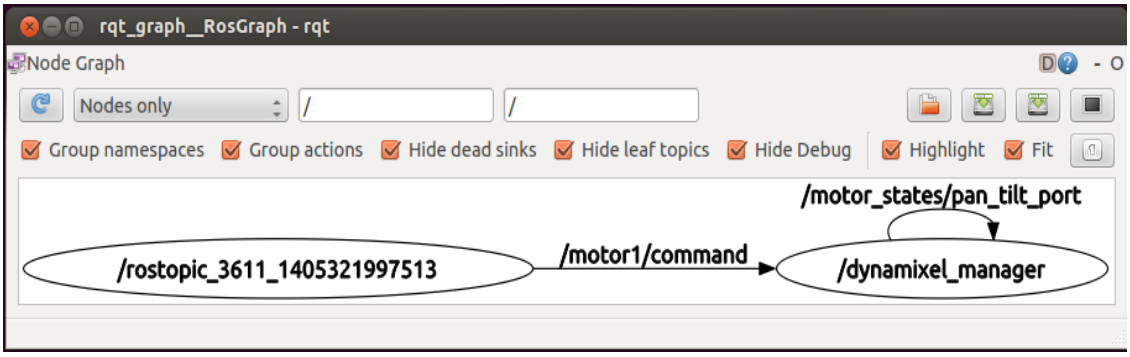

Figure 5.6. rqt_graph of the published topic.

5.2.2. *Node To Operate Dynamixel Servo:*

In the previous section we used the command rostopic pub to publish the angular position of a servo on a topic /tilt_controller/command for a single servo. Now in this section we are going to create a node to publish the data to this servo on the same topic.

This node gives input to the motor from -150 degrees to 150 degrees and then from 150 degrees to -150 degrees continuously. First servo moves from -150 to 150 degrees in CCW direction and then 150 to -150 degrees in CW direction. And this rotation happens continuously until we shutdown the node.

In this code initially the motor takes the position of -150 degrees. Then the motor reaches 150 degree position in CCW direction with an increment of one degree. Once the motor reaches the 150th degree position then it reverses its sense of rotation to CW direction to move from 150th degree position to -150 degree position and then it continues rotating by altering its sense of rotation. Here one should note that the publishing rate should match with the speed with which the servo is initialized.

The code of the node is as shown in below

publisher_continuous_motion.py

```
#!/usr/bin/env python
import roslib;roslib.load_manifest('dynamixel_tutorials')
import rospy
from std_msgs.msg import Float64

if __name__ =="__main__":
        rospy.init_node("Publisher")
        pub = rospy.Publisher('/motor1/command', Float64)
        r = rospy.Rate(40)
        if not rospy.is_shutdown():
                counter = -150
                i=0
                while (True):
                        int = counter*3.14/180  #Converting degrees to radians.
                        pub.publish(int)
        #This part ensures that the input is sent continously clockwise and anti-clockwise.
                        if(counter == 150):
                                i=1
                        elif(counter == -150):
                                i=0
                        if(i==1):
                                counter-=1
                        elif(i==0):
                                counter+=1
                        r.sleep()
```

Figure 5.7. A publisher node for continuous rotation.

5.2.3. *Node To Get The State of Dynamixel Servo:*

Till now we are only publishing the data to the servo on a topic /motor1/command. In this section we will create a node to get the state of the moving servo. The state of the servo implies the current position of the shaft, temperature inside the servo, information whether the servo shaft is moving or not, velocity of the shaft, load on the shaft, goal position, error in attaining the goal position, motor ID's etc.

The node to subscribe the state of the motor is shown in below

```
#!/usr/bin/env python
import roslib;roslib.load_manifest('dynamixel_tutorials')
import rospy
from dynamixel_msgs.msg import JointState

def callback(data):
        print data,"\n"
if __name__ == "__main__":
        rospy.init_node("Subscriber")
        rospy.Subscriber("/motor1/state", JointState, callback)
        rospy.spin()
```

Figure 5.8. A node to get state of the motor.

In the above code we created a node named "Subscriber". This node subscribe the state of the motor on a topic /motor1/state.

The computation graph when publisher and subscriber are executed is as shown

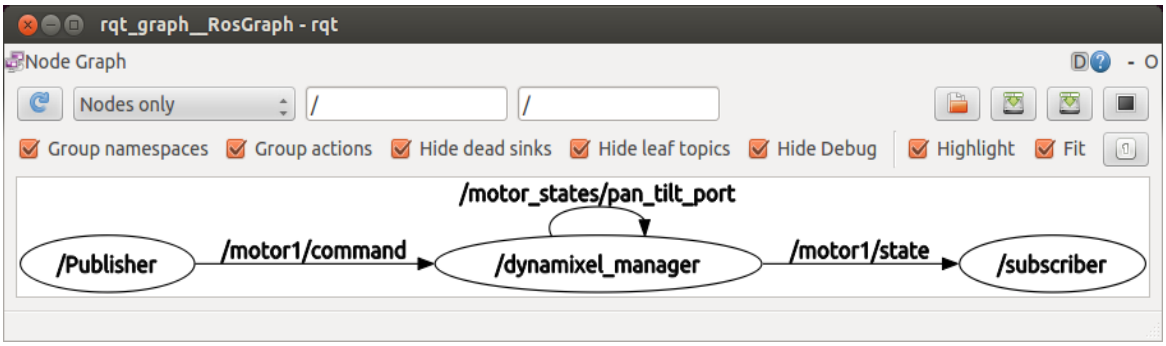

Figure 5.9. rqt_graph of the publisher and subscriber.

5.2.4. *Node To Operate One Servo By Taking Feedback From Other Servo:*

For this task we have connected two servos in daisy chain connection. First we should see whether the servos have unique ID's and same baud rate. Generally the default ID value of a servo is 1. So to make the ID's distinct from one another we need to change the ID of one servo from ID 1 to ID 2. This can done with the help of dynamixel driver package.

In this dynamixel_driver package there is a node named "change_id.py". We can change the ID of the servo with the help of following command.

$ rosrun dynamixel_driver change_id.py –port=/dev/ttyUSB0 –baud=1000000 [old id] [new id]

To set the same baud rate for two servos we need to run the node "set_servo_config.py" which is in dynamixel_driver package.

Use the following command to set the baud rate

$ rosrun dynamixel_driver set_servo_config.py –port=/dev/ttyUSB0 –baud=57600 –baud-rate =1 [motor ID1] [motor ID2] etc……

To configure two servos at a time we need to make some modifications in controller_spawner.launch file and dynamixel_joint_controllers.yaml file.

controller_spawner_modified.launch

```
<!-- -*- mode: XML -*- -->

<launch>
    <!-- Load controller configuration to parameter server -->
    <rosparam file="$(find dynamixel_tutorials)/config/dynamixel_joint_controllers.yaml" command="load"/>

    <!-- start specified joint controllers -->
    <node name="dynamixel_controller_spawner" pkg="dynamixel_controllers" type="controller_spawner.py"
          args="--manager=dxl_manager
                --port=pan_tilt_port
                --type=simple
                motor1
                motor2"
          output="screen"/>
</launch>
```

Figure 5.10. Modified controller_spawner.launch.

dynamixel_joint_controllers.yaml

```
motor1:
     controller:
          package: dynamixel_controllers
          module: joint_position_controller
          type: JointPositionController
     joint_name: tilt_joint
     joint_speed: 0.5
     torque_limit: 1
     torque: 1023
     motor:
          id: 1
          init: 512
          min: 0
          max: 1023

motor2:
     controller:
          package: dynamixel_controllers
          module: joint_position_controller
          type: JointPositionController
     joint_name: tilt_joint
     joint_speed: 0.5
     torque_limit: 1
     torque: 1023
     motor:
          id: 2
          init: 512
          min: 0
          max: 1023
```

Figure 5.11. Modified dynamixel_joint_controllers.yaml.

The code for the node to take feedback from servo with ID 1 and give this feedback to the servo with ID 2 is shown in below

Feedback.py

```
#!/usr/bin/env python
import roslib;roslib.load_manifest('dynamixel_tutorials')
import rospy
from dynamixel_msgs.msg import JointState
from std_msgs.msg import Float64

i=0
def callback(data):
        global i
        deg = int(data.current_pos*180/3.14)
        if( deg==149 or i==1):
                pub = rospy.Publisher("/motor2/command", Float64)
                pub.publish(float(data.current_pos))
                i=1
if __name__ == "__main__":
        rospy.init_node("Feedback")
        #i=0
        rospy.Subscriber("/motor1/state", JointState, callback)
        rospy.spin()
```

Figure 5.12. Feedback node.

From the servo with ID 1 feedback node takes the information. So first we need to move the shaft of the servo by publishing data to the servo with ID1 using publisher node. Once the servo 1 reaches a position of 150th degree, the feedback node starts publishing data to servo 2.

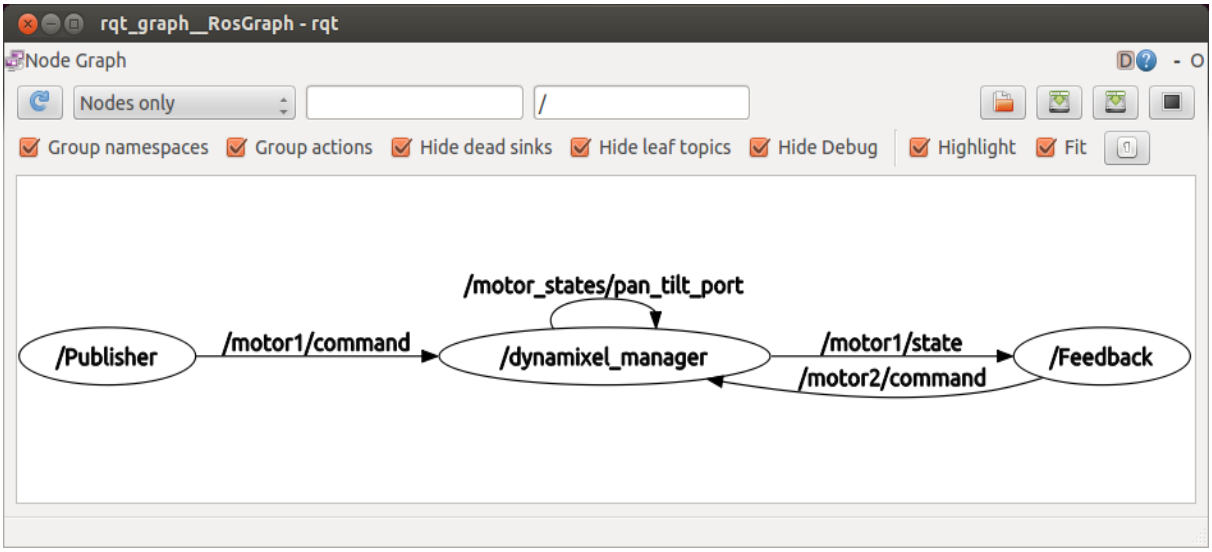

Figure 5.13. rqt_graph of Feedback node.

5.3. CREATING MIRROR MOTION OF ARMS

In this section we need to create a mirror motion between two arms. To do this task we need to establish a communication among 8 servos, with 4 servos on each arm. We need to move an arm manually and the other arm should follow this arm. For configuring 8 motors at a time we need to modify the spawner and yaml file.

In the below yaml file motor 2, motor 3, motor 4 namespaces have the same format of motor 1 with their respective ID's. Similarly motor 8, motor 9, motor 10 have the same format of motor 7.

controller_spawner_modified.launch

```
<!-- -*- mode: XML -*- -->

<launch>
    <!-- Load controller configuration to parameter server -->
    <rosparam file="$(find dynamixel_tutorials)/config/dynamixel_joint_controllers.yaml" command="load"/>

    <!-- start specified joint controllers -->
    <node name="dynamixel_controller_spawner" pkg="dynamixel_controllers" type="controller_spawner.py"
          args="--manager=dxl_manager
                --port=pan_tilt_port
                --type=simple
                motor1
                motor2
                motor3
                motor4
                motor7
                motor8
                motor9
                motor10"
          output="screen"/>
</launch>
```

Figure 5.14. Modified controller_spawner.launch for mirror motion.

dynamixel_joint_controllers.yaml

```
motor1:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: tilt_joint
    joint_speed: 0.5
    torque_limit: 1
    torque: 1023
    motor:
        id: 1
        init: 512
        min: 0
        max: 1023
...motor2, motor3, motor4
motor7:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: tilt_joint
    joint_speed: 0.5
    torque_limit: 0
    torque: 0
    motor:
        id: 7
        init: 512
        min: 0
        max: 1023
...motor8, motor9, motor10
```

Figure 5.15. Modified dynamixel_joint_controllers.yaml for mirror motion.

In order to move the arm manually we need to switch off the torque of motors 7, 8, 9, 10. To do this we will use the set_torque.py node in dynamixel_driver package. The command is shown below

```
$ rosrun dynamixel_driver set_torque.py –port=/dev/ttyUSB0 –baud=1000000 [motor ID] off
```

The node used to achieve this mirror motion is

Mirror.py

```
#! /usr/bin/env python
import roslib;roslib.load_manifest("dynamixel_tutorials")
import rospy
from std_msgs.msg import Float64
from dynamixel_msgs.msg import JointState

def callback(data):
        int=data.current_pos
        pub=rospy.Publisher("command", Float64)
        pub.publish(-int)

if(__name__ == '__main__'):
        rospy.init_node("pub_sub")
        rospy.Subscriber("state", JointState, callback)
        rospy.spin()
```

Figure 5.16. A mimic node to achieve mirror motion.

pub_sub is a mimic node. It subscribes to the topic "state" and publishes the data received from the "state" to the topic "command". In case of mismatch between servo angular positions of the two arms care should be taken to send the proportionate data. This can also be implemented on the leg of the humanoid.

Using launch file:

A launch file is used to call the above node with different name spaces to take data from one servo and send this data to the corresponding servo on the other arm.

The launch file is

Mirror.launch

```
<launch>
<group ns="motor7">
      <node name="pub_sub7" pkg="dynamixel_tutorials" type="Mirror.py">
      <remap from="command" to="/motor1/command"/>
      </node>
</group>
<group ns="motor8">
      <node name="pub_sub8" pkg="dynamixel_tutorials" type="Mirror.py">
      <remap from="command" to="/motor2/command"/>
      </node>
</group>
<group ns="motor9">
      <node name="pub_sub9" pkg="dynamixel_tutorials" type="Mirror.py">
      <remap from="command" to="/motor3/command"/>
      </node>
</group>
<group ns="motor10">
      <node name="pub_sub10" pkg="dynamixel_tutorials" type="Mirror.py">
      <remap from="command" to="/motor4/command"/>
      </node>
</group>
</launch>
```

Figure 5.17. A launch file for mirror motion.

In this launch file the topic command is remapped to /motor1/command, /motor2/command, /motor3/command and /motor4/command to send the data received to the respective topics.

5.4. PYTHON LIBRARY

There are python packages available to work with Excel files that will run on any Python platform and that do not require either Windows or Excel to be used. xlrd is a python library to extract data from Microsoft Excel spreadsheet files. xlrd can extract data from Excel spreadsheets (.xls and .xlsx, versions 2.0 onwards) on any platform, is written in pure Python and is Unicode-aware.

The package itself is pure Python with no dependencies on modules or packages outside the standard Python distribution. xlrd provides access to named constants and named groups of cells. It also provides access to "visual" information: font, "number format", background, border, alignment and protection for cells, height/width etc for rows/columns.

Example:

1. Grab a specific worksheet from a workbook:

```
import xlrd
workbook = xlrd.open_workbook('my_workbook.xls')
worksheet = workbook.sheet_by_name('Sheet1')
```

2. Iterate over each worksheet in a workbook:

```
import xlrd
workbook = xlrd.open_workbook('my_workbook.xls')
worksheets = workbook.sheet_names()
for worksheet_name in worksheets:
        worksheet = workbook.sheet_by_name(worksheet_name)
        print worksheet
```

3. Iterate over each row of a worksheet:

```
import xlrd
workbook = xlrd.open_workbook('my_workbook.xls')
worksheet = workbook.sheet_by_name('Sheet1')
num_rows = worksheet.nrows - 1
curr_row = -1
while curr_row < num_rows:
        curr_row += 1
        row = worksheet.row(curr_row)
        print row
```

4. Grab the cell contents of each row of a worksheet:

```
import xlrd
workbook = xlrd.open_workbook('my_workbook.xls')
worksheet = workbook.sheet_by_name('Sheet1')
num_rows = worksheet.nrows - 1
num_cells = worksheet.ncols - 1
curr_row = -1
while curr_row < num_rows:
        curr_row += 1
        row = worksheet.row(curr_row)
        print 'Row:', curr_row
        curr_cell = -1
        while curr_cell < num_cells:
                curr_cell += 1
                # Cell Types: 0=Empty, 1=Text, 2=Number, 3=Date, 4=Boolean, 5=Error, 6=Blank
                cell_type = worksheet.cell_type(curr_row, curr_cell)
                cell_value = worksheet.cell_value(curr_row, curr_cell)
                print ' ', cell_type, ':', cell_value
```

Similar to the xlrd package that only reads an excel spreadsheet file there is a package named xlwt to write data into the excel spreadsheet file.

5. Simple example to generate an excel sheet

```
import xlwt
workbook = xlwt.Workbook()
worksheet = workbook.add_sheet('My Worksheet')
worksheet.write(0, 0, label='Row 0,Column 0 Value')
workbook.save('Excel_workbook.xls')
```

5.5. NODE TO SEND DATA THROUGH EXCEL SHEET

In this section we need to create a node to take data from an excel sheet. This excel sheet contains the information of the angular positions of the servos.

Here we create a node to take the data from the excel sheet that contains the angular positions of the servos in radians and publish them to the respective servos.

The data in the excel sheet is as follows,

	1	2	3	4
2	0	0	0	0
3	0.5	0.5	0.5	0.5
4	0	0	1	1
5	-0.5	-0.5	-1	-1
6	0	0	-0.5	-0.5
7	0	0	0	0
8				

Figure 5.18. Excel sheet data in radians.

The first column contains the data for servo 1, second for servo 2 and so on.

The node to achieve this is as shown below,

Publisherxls.py

```
#!/usr/bin/env python
import roslib;roslib.load_manifest('dynamixel_tutorials')
import rospy
from std_msgs.msg import Float64
import xlrd

def publish(data, topic, r):
        pub = rospy.Publisher("/motor"+topic+"/command", Float64)
        int=data
        pub.publish(int)
        r.sleep()

if __name__ =="__main__":
        rospy.init_node("Publisher")
        r = rospy.Rate(10) #Traverse the loop 10 times in a second.
        workbook = xlrd.open_workbook('/home/sampath/Desktop/sample.xls')
        worksheet = workbook.sheet_by_name('Data')
        numrows = worksheet.nrows
        curr_row = 0
        while ((curr_row<numrows) and (not rospy.is_shutdown())):
                for i in range(4):
                        publish(worksheet.cell_value(curr_row, i), str(i+1), r)
                curr_row+=1
```

Figure 5.19. A node to take data from excel sheet data in radians.

The excel sheet may also contain the coordinates of specified points on human body. This sort of excel is generated in MATLAB environment. A gait is captured by a high definition camera and stored as a video. This video is divided into number of frames. In each frame the

coordinates of the specified points are generated. These coordinates of each frame are stored in an excel sheet.

The node takes the coordinates of specified points of different frames, compute the absolute angle by choosing the appropriate reference for each link with the help of trigonometric relations. This angle is now sent as an input to the respective servos.

The node that does this operation is as shown below,

publisher_motion_detector_angle.py

```
#!/usr/bin/env python
# importing various packages
import roslib;roslib.load_manifest('dynamixel_tutorials')
import rospy, xlrd, math
from std_msgs.msg import Float64

# Function to publish data to different specified motors

def publish(data, number, r):
        pub = rospy.Publisher("/motor"+number+"/command", Float64)
        int=data
        pub.publish(int)
        r.sleep()

# Function that returns the distance between two points when their coordinates are sent
as arguments

def point_sum(x1,x2,y1,y2):
        return math.sqrt(math.pow((x2-x1),2) + math.pow((y2-y1), 2))

# Calculates the slope for the line with specified points

def slope(x1,x2,y1,y2):
        if x2==x1:
                return 0
        else:
                return (y2-y1)/(x2-x1)

if (__name__ =="__main__" and not rospy.is_shutdown()):
        rospy.init_node("Publisher")
        x = rospy.Rate(2)
        wb = xlrd.open_workbook('/home/sampath/Desktop/New folder1/output.xls')
        ws = wb.sheet_by_index(0)
        numrows = ws.nrows
```

```
        f=open('/home/sampath/Desktop/data1.txt','w')
        # A frame contains i-1 points.
        for i in range(1,numrows):
                if(ws.cell_value(i,1) == 0):
                        break
        a=list() # List of motor id's
        b=list() # List of inputs to the respective motors.
        for k in range(1,i-2):
                z=raw_input("Enter id's of motors: ")
                a.append(str(z))
                b.append(0)
        j=0
        curr_row=1
        while (curr_row<(numrows-2) and not rospy.is_shutdown()):
                if not (ws.cell_value(curr_row, 1)== 0 or ws.cell_value(curr_row+1, 1)
== 0 or ws.cell_value(curr_row+2, 1)==0):
                        s1=point_sum(ws.cell_value(curr_row,0),ws.cell_value(curr_row
+1,0),ws.cell_value(curr_row,1),ws.cell_value(curr_row+1,1))
                        s2=point_sum(ws.cell_value(curr_row+1,0),ws.cell_value(curr_row
+2,0),ws.cell_value(curr_row+1,1),ws.cell_value(curr_row+2,1))
                        s3=point_sum(ws.cell_value(curr_row,0),ws.cell_value(curr_row
+2,0),ws.cell_value(curr_row,1),ws.cell_value(curr_row+2,1))
                        s4=math.pow(s1,2)+math.pow(s2,2)-math.pow(s3,2)
                        s5=float(2*s1*s2)
                        m1=slope(ws.cell_value(curr_row,0),ws.cell_value(curr_row
+1,0),ws.cell_value(curr_row,1),ws.cell_value(curr_row+1,1))
                        m2=slope(ws.cell_value(curr_row,0),ws.cell_value(curr_row
+2,0),ws.cell_value(curr_row,1),ws.cell_value(curr_row+2,1))
                        theta = 3.14-math.acos(s4/s5) if m2>m1 else math.acos(s4/
s5)-3.14
                        if (math.copysign(theta-b[j], 1) > 0.29): # To ensure that
input difference is upto the resolution.
                                publish(theta, a[j], x)
                                b[j] = theta
                                f.write(str(b[j])+'\tmotor'+a[j]+'\n') # creates a file
with sent inputs for specific motors.
                j=0 if (ws.cell_value(curr_row, 1) == 0) else j+1
                curr_row+=1
f.close()
```

Figure 5.20. A node to take data from excel sheet data in coordinates of specified points.

If in case the servo is fitted in a way that the angular positions of the servos do not coincide with zero for the reference line taken, then we need to maintain a separate list of initial positions of the servos and we need to add them to the calculated theta

The angle is calculated using a trigonometric relation,

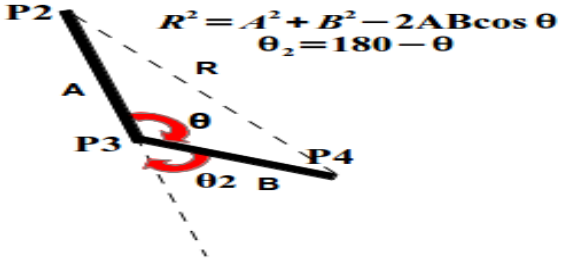

Figure 5.21.Trigonometric relation illustration

The reference is taken as shown below,

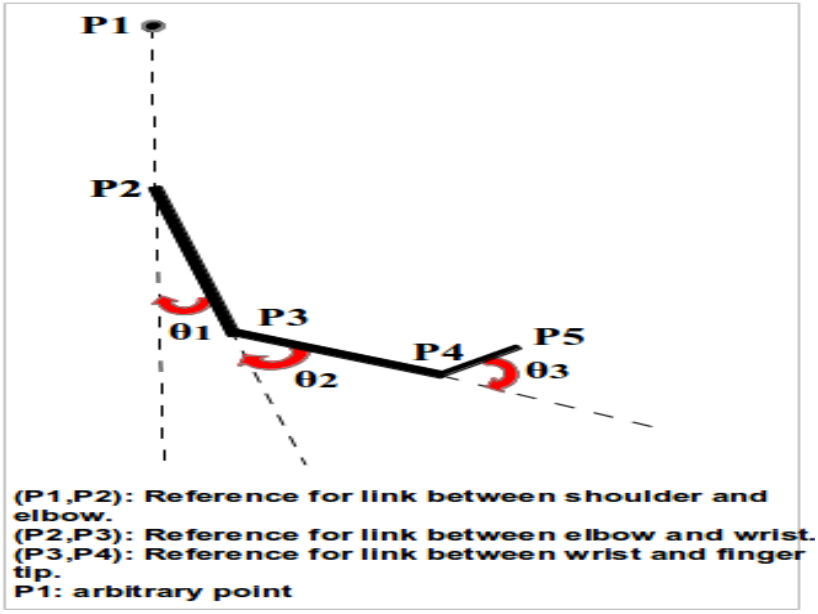

Figure 5.22.References taken for the hand of humanoid

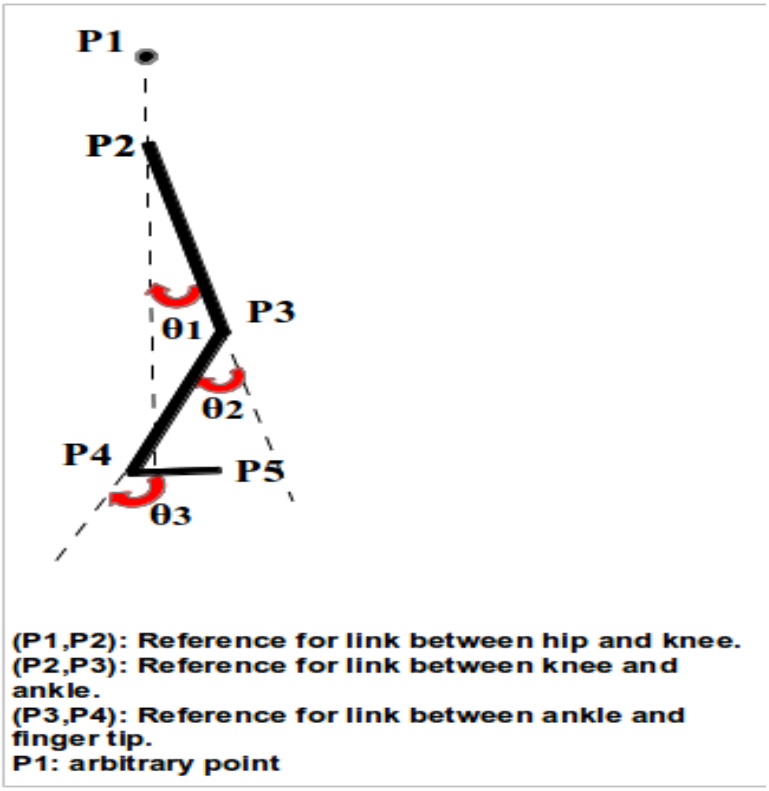

Figure 5.23.References taken for the leg of humanoid

Chapter6:

RESULTS AND DISCUSSIONS

6.1. IMITATION

The result of the task 3 i.e., the movement of one arm of IIT Kharagpur humanoid manually, leads the other arm to automatically move is depicted in the following graph

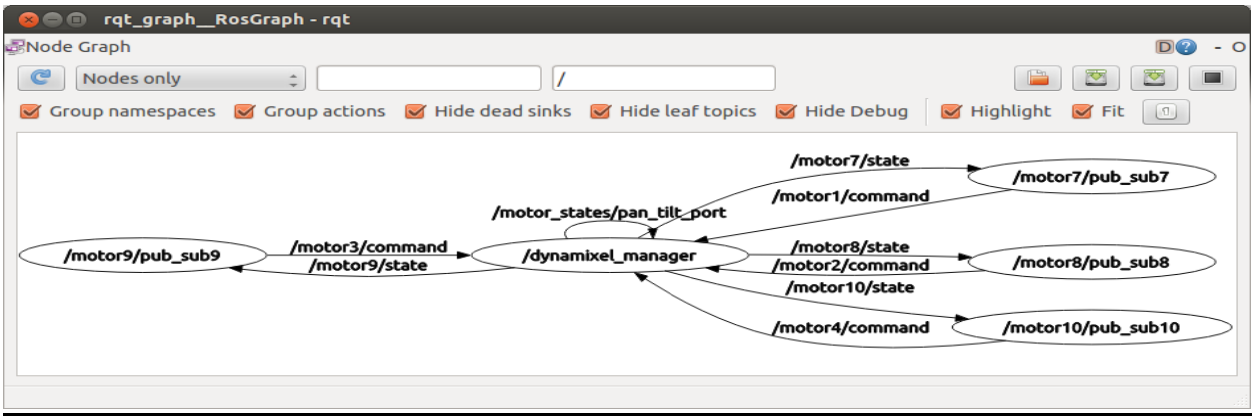

Figure 6.1. rqt_graph of mirror motion.

We assigned the ID's 7, 8, 9 and 10 to servos of an arm that is manually moved. To the servos of the other arm of the humanoid we assigned ID's 1, 2, 3 and 4. In the above graph there are five nodes running at a time. Among which four nodes are subscribing the state of servos with ID's 7, 8, 9 and 10 from dynamixel_manager node and publish this subscribed angular position of the servos to servos with ID's 1, 2, 3 and 4 respectively. By doing so we achieved the imitation of the arm that is moved manually. The same can be done to the legs of the humanoid.

The videos for this result are available in the following web link.

http://youtu.be/1G29jx9LYI8

6.2. INTEGRATION

We are able to integrate the data that is generated in the mat lab environment in the form of an excel sheet that contains the coordinates of specified points on a moving object for each frame, with the ROS environment where in we read the coordinate data from the file and convert this data into angles and these angles that are in radians were sent as an input to the dynamixel servos at particular joints to resemble the moving object. A node can also be written to get the

error in achieving the goal position from every motor. So that we can determine the consistency of integration.

The flowchart of events is as shown below.

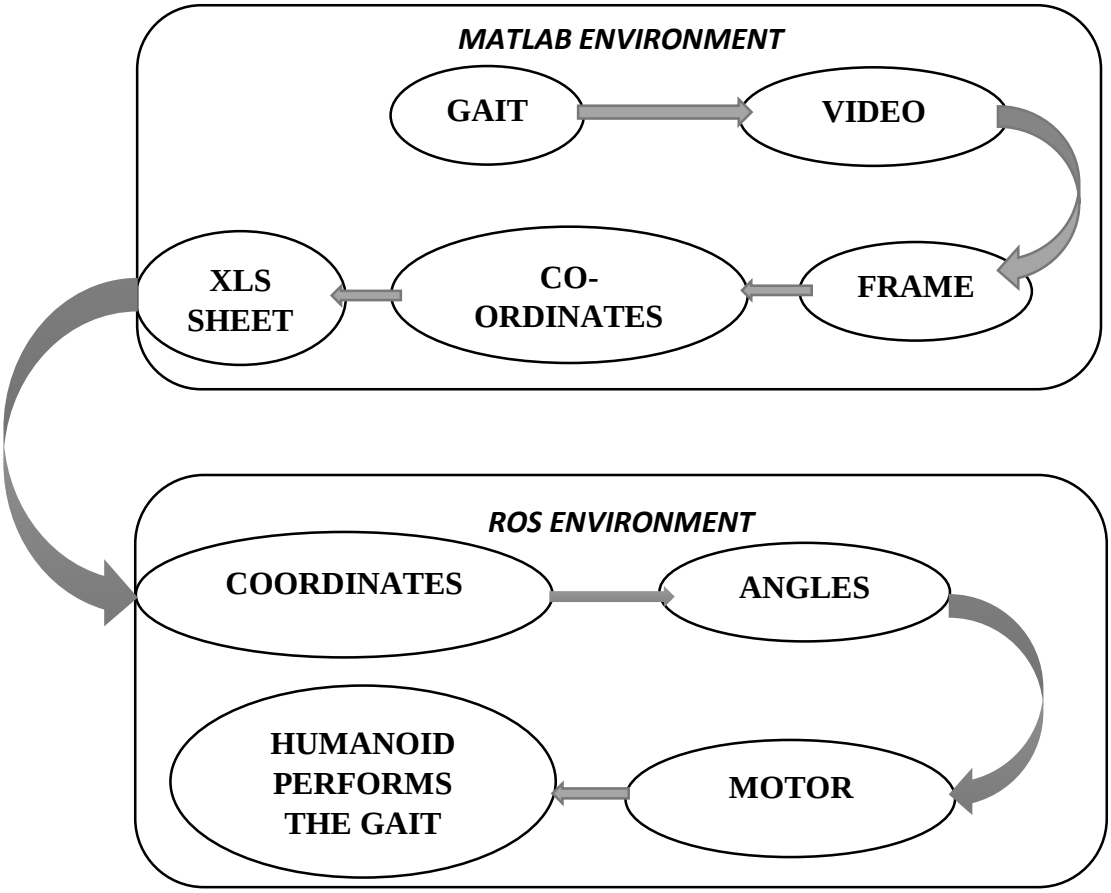

Figure 6.2. Flow chart of events in integration.

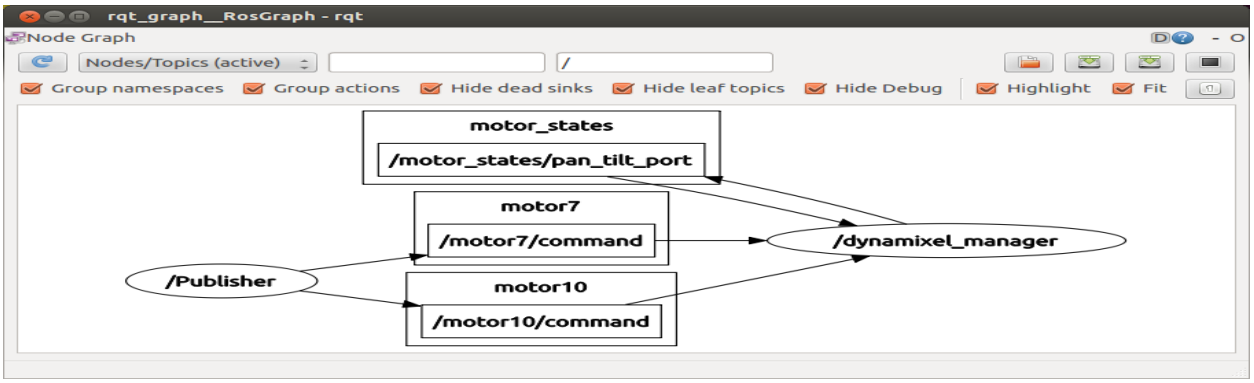

Figure 6.3. rqt_graph for resembling a gait.

The videos for this result are available in the following web links.

Tracking	Actual motion	Humanoid motion
Absolute angle	http://youtu.be/wQdQUjCVHVA	http://youtu.be/H8GUGIFow8Q
Hand motion1	http://youtu.be/eImuPikdzpg	http://youtu.be/RFlepOL4LOM
Hand motion2	http://youtu.be/VZFaTO52O8Q	http://youtu.be/S687munfx9c
Hand motion3	http://youtu.be/9iQFQNOClyA	http://youtu.be/6z_nBEK9kSs

Table 6.1. Web links for the imitation of human by humanoid

6.3. ERRORS AND DISCUSSIONS

- **Installation error**

While installing ROS on UBUNTU we may face an error showing broken package dependencies if ROS is not completely installed.

The solution to this problem is to run the installation command again so that the packages that are failed to be downloaded will be downloaded this time.

- **Master not found**

When we execute a node without using launch file we may find an error 'master not found.'

This problem can be overcome by running roscore in a new shell which starts master for the nodes to register.

- **Executable error**

When we execute a new node we may find an error 'not executable.'

To avoid this we should make the node as an executable one by running the command

```
$ chomod +x [/path/node_name]
```

- **Permission denied error**

When we connect USB2Dynamixel to our PC for the first time the controller_manager may not have permission to access the port.

The solution to this is to go to directory named dev in filesystem and look for the file named ttyUSB0 and change the permissions for the file to be accessible by the user.

- **No motor found error**

Some times controller_manager may show the error ‘no motors found’. This is due to the following reasons.

a. Wrong connections

b. Having baud rate, ID different from specified values.

If we give wrong connections to the motors, the USB2Dynamixel will glow continuously even before connecting it to the PC. If this is so, connect the motors properly.

If the motors have different baud rates and same ID, set them to the appropriate values using dynamixel_driver package.

Chapter7:

CONCLUSIONS AND SCOPE FOR THE FUTURE STUDY

7.1. IMPLIMENTATION OF GAITS

Simple motions of a human can be captured and this captured video can be further processed through MATLAB to get the coordinates of specific points. These coordinates can be further processed in ROS into angles, which are sent as an input to the specific motors. By doing so we can implement that captured motion on the humanoid. In the same way we can implement different sort of gaits.

7.2. GENERATION OF GAITS

After implementing different gaits on the humanoid we can make this humanoid to store these gaits and when required the humanoid could be made capable of generating its own gaits based on the previously stored gaits. This improves the autonomous level of the humanoid thereby enhance its artificial intelligence.

7.3. VIRTUAL CONTROL

For distant education or application we use virtual labs. From a distant place we can send input to the machines or robots that are in labs or fields. Using ROS the humanoid can also be controlled from a remote place. So, this can also be undertaken as a future scope of study to control the humanoid from a distant place.

REFERENCES:

Reference to Manual

ROBOTIS e-Manual v1.21.00 (2006), AX-12/ AX-12+/ AX-12A, Robotis, Korea, Asia.

Reference to Thesis

SAURYA PRAKASH MISHRA (2010), Implementation of Semi-Generative Gaits in an Adult Size Humanoid Robot, IIT KGP, West Bengal ,India.

Reference to Text book

Aaron Martinez, Enrique Fernández (2013), Learning ROS for Robotics Programming, Packt Publishing Company, Birmingham, UK.

Reference to Web

[1] 213.253.39.173 (talk), Toronto30(talk |contribs), 15:08,15July2014, Robotics, http://en.wikipedia.org / wiki/Robotics

[2] Galileo education network, GENA, (1999-2003), Introduction To Robots http://www.galileo.org/robotics/intro.html

[3] Brian Dunbar, David Hitt (September 6th, 2013), What Is Robotics? NASA Educational Technology Services, http://www.nasa.gov/audience/foreducators/robotics/home/what_is_robotics_k4.html#.U7vgqaMQad9

[4] Thomas Jaspers (March 23, 2012), Robot Framework Tutorial https://blog.codecentric.de/en/2012/03/robot-framework-tutorial-overview/

[5] http://robotframework.org

[6] Montrealais (01:22, 1 September 2003), Humanoid, http://en.wikipedia.org/wiki/Humanoid

[7] Saurya Mishra(2012), The KYZR Humanoid Project, https://sites.google.com/site/kyzriitkgp/project-updates

[8] Saurya Mishra(2012), The KYZR Humanoid Project, https://sites.google.com/site/kyzriitkgp/

[9] Robotisinc (01:30, 28 December 2011), DYNAMIXEL, http://en.wikipedia.org/wiki/DYNAMIXEL

[10] John Machin (2014/4/9),XLRD 0.9.3, http://pypi.python.org/pypi/xlrd

[11] SJ Machin, The xlrd Module, http://www.lexicon.net/sjmachin/xlrd.htm

[12] SJ Machin, http://www.lexicon.net/sjmachin/README.html

[13] Creative Commons Attribution 3.0(2014), ROS, http://www.ros.org

[14] vibrunazo(last edited 2014-06-22 06:22:25), http://wiki.ros.org/ROS/Tutorials

[15] Antons Rebguns(2014), ROS stack for interfacing with Robotis Dynamixel line of servo motors, https://github.com/arebgun/dynamixel_motor

Appendix I:

Acceptance Test-Driven Development (ATDD) is a development methodology based on communication between the business customers, the developers, and the testers. ATDD encompasses many of the same practices as Specification by Example, Behavior Driven Development (BDD), Example-Driven Development (EDD), and Story Test-Driven Development (SDD). All these processes aid developers and testers in understanding the customer's needs prior to implementation and allow customers to be able to converse in their own domain language.

ATDD is closely related to Test-Driven Development [TDD]. It differs by the emphasis on developer-tester-business customer collaboration. ATDD encompasses, but highlights writing acceptance tests before developers begin coding.