RAL 333 ROBOT OPERATING SYSTEM LAB

LAB RECORD

ROBOTICS AND AUTOMATION 5th SEMESTER

Toc H INSTITUTE OF SCIENCE & TECHNOLOGY
Arakkunnam P.O, Ernakulam District, Kerala – 682 313.

VISION OF THE INSTITUTION

To become a globally recognized institution that develops professionals with integrity who excel in their chosen domain making a positive impact in industry, research, business and society.

MISSION OF THE INSTITUTION:

- To provide the ambience necessary to achieve professional and technological ex-cellence at the global level.
- To undertake collaborative research that fosters new ideas for sustainable development.
- To instill in our graduates' ethical values and empathy for the needs of society

VISION OF THE DEPARTMENT

"To achieve excellence at the global level in Robotics & Automation, fostering in our students technological proficiency, interdisciplinary approach, innovation, professional and social commitment"

MISSION OF THE DEPARTMENT

- To impart strong fundamentals in Robotics & Automation through project based approach and in collaboration with industries.
- To establish state of the art laboratories to facilitate research and innovation
- To continuously upgrade the knowledge and skills of faculty to incorporate the latest advancements
- To engage the students in research through investigation and critical thinking with awareness of societal and ecological considerations
- To inculcate effective communication skills, ethical practices and professional skills to work in a collaborative environment

PROGRAMME EDUCATIONAL OBJECTIVES (PEO)

The graduates will

- Demonstrate technical competence in identifying, analyzing and designing innovative, sustainable and cost effective solutions for complex problems in multidisciplinary fields.
- Adapt to rapid changes in tools and technology with an understanding of societal and ecological issues relevant to professional engineering practice through life-long learning.
- Work successfully in collaborative and multidisciplinary environments upholding professional and ethical values

PROGRAM SPECIFIC OUTCOMES

A graduate of the Robotics & Automation program will demonstrate:

PSO1: Professional skills: An ability to apply concepts in Robotics & Automation to design and implement complex machines and systems to solve problems faced in manufacturing, health care, agriculture, industrial engineering and safety.

PSO2: Competitive skills: An ability to make use of acquired technical knowledge for successful career, to align with changing industry requirements and qualifying in competitive examinations

Toc H INSTITUTE OF SCIENCE & TECHNOLOGY
Arakkunnam P.O, Ernakulam District, KERALA – 682 313

DEPARTMENT OF ROBOTICS AND AUTOMATION

CERTIFICATE

This is to certify that this is the bonafide record of practical work done by.....who has satisfactorily completed the course of practicals described by APJ Abdul Kalam Technological University in the RAL333 Robot Operating System laboratory of Toc H Institute of Science & Technology Arakkunnam during the academic year ……

Register Number :

Branch :

Staff in charge **Head of the Department**

Submitted for the FIFTH semester B.Tech degree practical examination held at Toc H Institute of Science & Technology on …….

Internal Examiner **External Examiner**

Course Code: RAL 333

Course Name: ROBOT OPERATING SYSTEM LAB

Course Outcomes: After the completion of the course the student will be able to

CO1- Analyze the applications of ROS in real world complex scenarios
CO2- Explain the usage of turtlesim and Rviz in ROS
CO3- Explain the use of G a z e b o and MoveIt in ROS
CO4- Describe about the concepts behind navigation
CO5- Analyze the issues in hardware interfacing with ROS environment

Course Outcomes	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	2	2	2	3	2			2	2		3
CO2	3	2	2	2	3	2			2	2		3
CO3	3	2	2	2	3	2			2	2		3
CO4	3	2	2	2	3	2			2	2		3
CO5	3	2	2	2	3	2			2	2		3

Course Outcomes	PSO1	PSO2
CO1	3	3
CO2	3	3
CO3	3	3
CO4	3	3
CO5	3	3

SL.NO.	LAB EXPERIMENT	COs & PO/PSOs MAPPING
	PARTA	
1	Introduction to Robot Operating System (ROS)	CO1, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1, PSO2
2	Writing a Simple Publisher and Subscriber, Simple Service and Client, Recording and playing back data, Reading messages from a bag file(Python/C++)	CO1, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1, PSO2
3	Getting Started with Turtlesim	CO1, CO2, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1, PSO2
4	Familiarisation with Rviz -- Markers: Sending Basic Shapes -- use visualization_msgs/Marker messages to send basic shapes, to send points and lines (C++), Interactive Markers: Writing a Simple Interactive Marker Server, Basic Controls	CO1, CO2, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1, PSO2
5	Introduction to tf -- broadcast the state of a robot to tf, get access to frame transformations, adding a frame, wait For Transform function, setting up your robot using tf, publish the state of your robot to tf, using the robot state publisher.	CO1, CO2, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1, PSO2
6	Building a Visual Robot Model with URDF from Scratch, building a Movable Robot Model with URDF, Adding Physical and Collision Properties to a URDF Model.	CO1, CO2, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12,PSO1, PSO2
7	Familiarisation with Gazebo--How to get Gazebo up and running, Creating and Spawning Custom URDF Objects in Simulation, Gazebo ROS API for C-Turtle, Simulate a Spinning Top, Gazebo Plugin - how to create a gazebo plugin, Create a Gazebo Plugin that Talks to ROS	CO1, CO3 PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1,PSO2
8	Create a Gazebo Custom World (Building Editor, Gazebo 3D Models), Add Sensor plugins like Laser, Kinect, etc. to URDF of mobile robot	CO1,CO3,PO1, PO2, PO3, PO4,PO5,PO6,PO9,PO10, PO12,PSO1,PSO2
9	Create a 3DOF robotic arm from scratch	CO1,CO2,CO3,PO1,PO2, PO3,PO4,PO5,PO6,PO9, PO10,PO12,PS01,PS02

10	Create Moveit package for robotic arm simulation and add controllers, Plan a path for a 3DOF Robotic Arm and execute the same, Move the 3DOF arm to a desiredgoal point	CO1,CO2,CO3,PO1, PO2, PO3, PO4, PO5, PO6,PO9,PO10,PO12, PSO1,PSO2
11	Execute SLAM Mapping (Lidar based) using a differentially driven mobile robot	CO1,CO3,CO4,PO1, PO2,PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1, PSO2
PARTB		
12	Familiarise ROS Serial Arduino for hardware interface.	CO1,CO5, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1, PSO2
ADDITIONAL EXPERIMENTS		
13	Creating and Spawning Custom URDF Objects in Simulation	CO1, CO2, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12,PSO1, PSO2
14	Setting up your robot using tf	CO1, CO2, PO1, PO2, PO3, PO4, PO5, PO6, PO9, PO10, PO12, PSO1, PSO2

Table of Contents

SL.NO.	DATE	LAB EXPERIMENT	Page. No	SIGNATURE
		PART A		
1		Introduction to Robot Operating System (ROS)		
2		Writing a Simple Publisher and Subscriber, Simple Service and Client, Recording and playing back data, Reading messages from a bag file(Python/C++)		
3		Getting Started with Turtlesim		
4		Familiarisation with Rviz -- Markers: Sending Basic Shapes -- use visualization_msgs/Marker messages to send basic shapes, to send points and lines(C++), Interactive Markers: Writing a Simple Interactive Marker Server, Basic Controls		
5		Introduction to tf -- broadcast the state of a robot to tf, get access to frame transformations, adding a frame, wait For Transform function, Setting up your robot using tf, publish the state of your robot to tf, using the robot state publisher.		
6		Building a Visual Robot Model with URDF from Scratch, building a Movable Robot Model with URDF, Adding Physical and Collision Properties to a URDF Model.		
7		Familiarisation with Gazebo--How to get Gazebo up and running, Creating and Spawning Custom URDF Objects in Simulation, Gazebo ROS API for C-Turtle, Simulate a Spinning Top, Gazebo Plugin - how to create a gazebo plugin, Create a Gazebo Plugin that Talks to ROS		
8		Create a Gazebo Custom World (Building Editor, Gazebo 3D Models), Add Sensor plugins like Laser, Kinect, etc. to URDF of mobile robot		
9		Create a 3DOF robotic arm from scratch		

10		Create Moveit package for robotic arm simulation and add controllers, Plan a path for a 3DOF Robotic Arm and execute the same, Move the 3DOF arm to a desiredgoal point		
11		Execute SLAM Mapping (Lidar based) us-ing a differentially driven mobile robot		
PART B				
12		Familiarise ROS Serial Arduino for hard-ware interface.		
ADDITIONAL EXPERIMENTS				
13		Creating and Spawning Custom URDF Objects in Simulation		
14		Setting up your robot using tf		

Introduction to Robot Operating System (ROS)

ROS is an open-source, meta-operating system for the robot. It provides the services that 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.

Goals of the ROS framework:

- Thin: ROS is designed to be as thin as possible
- ROS-agnostic libraries: the preferred development model is to write ROS-agnostic libraries with clean functional interfaces.
- Language independence: the ROS framework is easy to implement in any modern programming language.
- Easy testing: ROS has a built-in unit/integration test framework called ROS test that makes it easy to bring up and tear down test fixtures.
- Scaling: ROS is appropriate for large runtime systems and for large development processes.

Operating system

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

ROS Distributions

The core ROS system, along with useful tools and libraries are regularly released as a ROS Distribution. This distribution is similar to a Linux distribution and provides a set of compatible software for others to use and build upon.

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (**Recommended**)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level.

ROS Filesystem Level

The filesystem level concepts mainly cover ROS resources that encounter on disk, such as:

- **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 atomic build item and release item in ROS. Meaning that the most granular thing can build and release is a package.
- **Metapackages**: Metapackages are specialized Packages which only serve to represent a group of related other packages. Most commonly metapackages are used as a backwards compatible place holder for converted ROS build Stacks.

- **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. The package.xml package manifest is defined in REP-0127.
- **Repositories**: A collection of packages which share a common VCS system. Packages which share a VCS share the same version and can be released together using the catkin release automation tool bloom. Often these repositories will map to converted ROS build Stacks. Repositories can also contain only one package.
- **Message(msg) types**: Message descriptions, stored in my_package/msg/MyMessageType.msg, define the data structures for messages sent in ROS.
- **Service (srv) types**: Service descriptions, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS.

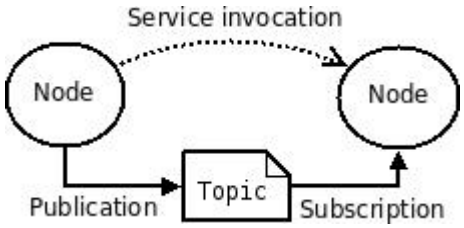
ROS Computation Graph Level

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. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser rangefinder, one node controls the wheel motors, one node performs localization, one node performs path planning, 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, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).
- **Topics**: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by *publishing* it to a given topic. 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. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each other's existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.
- **Services**: The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply to interactions, which are often required in a distributed system. 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. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.
- **Bags**: Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.



ROS_basic_concepts.dia

ROS Community Level

The ROS Community Level concepts are ROS resources that enable separate communities to exchange software and knowledge. These resources include:

- **Distributions**: ROS Distributions are collections of versioned stacks that you can install. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software.
- **Repositories**: ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.
- **The ROS Wiki**: The ROS community Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.
- **Bug Ticket System**: Please see Tickets for information about file tickets.
- **Mailing Lists**: The ros-users mailing list is the primary communication channel about new updates to ROS, as well as a forum to ask questions about ROS software.
- **ROS Answers**: A Q&A site for answering your ROS-related questions.
- **Blog**: The ros.org Blog provides regular updates, including photos and videos.

Catkin Workspace

- A *catkin workspace* is a set of directories in which a set of related ROS code/packages live (catkin ~ ROS build system: CMake + Python scripts)
- It's possible to have multiple workspaces, but work can performed on only one-at-a-time
- A ROS package is a directory inside a **catkin workspace** that has a *package.xml* file in it
- A package contains the source files for one node or more and configuration files

Catkin Workspace Folders

Source space	Contains the source code of catkin packages. Each folder within the source space contains one or more catkin packages.
Build Space	is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here.
Development (Devel) Space	is where built targets are placed prior to being installed
Install Space	Once targets are built, they can be installed into the install space by invoking the install target. 30

Installing ROS on Ubuntu

- Ubuntu Version :20.04 LTS 64 bit
- ROS Distribution: ROS Noetic Ninjemys

1) Download Ubuntu 20.04.3 from the following link

https://ubuntu.com/download/desktop

2) Download Virtual Box for Windows from the following link

https://www.virtualbox.org/wiki/Downloads

3) Install Ubuntu on Virtual Box using the following link

https://robocademy.com/2020/05/17/best-4-ways-to-install-ubuntu-for-ros/

4) Install ROS Noetic on Ubuntu using the following link

https://robocademy.com/2020/05/23/getting-started-with-new-ros-noetic-ninjemys/

EXPERIMENT 1

AIM: Writing a Simple Publisher and Subscriber, Simple Service and Client, Recording and playing back data.

1. Writing a Simple Publisher and Subscriber

1. Writing the Publisher Node

"Node" is the ROS term for an executable program that is connected to the ROS network. Here we'll create the publisher ("talker") node which will continually broadcast a message.

STEPS TO BE FOLLWED

1) Building a catkin workspace

   mkdir name of your workspace

2) Create src folder in the catkin workspace

   cd name of your workspace

   mkdir src

3) Build the packages in the catkin workspace:

   cd ~/ name of your workspace

   catkin_make

   (NB-This command should be applied staying in the catkin workspace)

4) To add the workspace to your ROS environment you need to source the generated setup file:

   source devel/setup.bash

   (NB-This command should be applied staying in the catkin workspace)

5) Creating a catkin Package

   cd ~/ name of your workspace /src

   catkin_create_pkg beginner_tutorials std_msgs rospy roscpp

   This will create a beginner_tutorials folder(package) which contains a package.xml and a CMakeLists.txt

6) Change directory into the beginner_tutorials package

   roscd beginner_tutorials

7) Create a 'scripts' folder to store our Python scripts

   mkdir scripts

   cd scripts

   gedit talker.py (gedit is a command used to create a text file

Copy the below python script to talker.py(text file created in the script folder)

talker.py

```
#!/usr/bin/env python3
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world"
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '_main_':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

8) Make the python file executable using the command

chmod +x talker.py (This code will make the program executable)

9) Building your node

This is to make sure that the autogenerated Python code for messages and services is created.

(Go to your catkin workspace and run catkin_make)

cd ~/ name of your workspace

catkin_make

10) . Running a publishing node

a. Run Ros master in a **new terminal 1**

roscore

b. To add the workspace to your ROS environment you need to source the generated setup file**:(Done in new terminal 2)**

cd ~/ name of your workspace

source devel/setup.bash

c. Run the publishing node

rosrun beginner_tutorials talker.py

You will see something similar to

```
[INFO] [1641920416.141848]: hello world
[INFO] [1641920416.241844]: hello world
[INFO] [1641920416.341824]: hello world
[INFO] [1641920416.441849]: hello world
[INFO] [1641920416.541808]: hello world
[INFO] [1641920416.641769]: hello world
[INFO] [1641920416.741781]: hello world
[INFO] [1641920416.841791]: hello world
[INFO] [1641920416.941768]: hello world
[INFO] [1641920417.041645]: hello world
[INFO] [1641920417.141843]: hello world
[INFO] [1641920417.241872]: hello world
[INFO] [1641920417.341849]: hello world
[INFO] [1641920417.441807]: hello world
[INFO] [1641920417.541849]: hello world
[INFO] [1641920417.641634]: hello world
[INFO] [1641920417.741840]: hello world
[INFO] [1641920417.841863]: hello world
[INFO] [1641920417.941855]: hello world
[INFO] [1641920418.041810]: hello world
[INFO] [1641920418.141853]: hello world
[INFO] [1641920418.241864]: hello world
[INFO] [1641920418.341841]: hello world
```

2. Writing the Subscriber Node

STEPS TO BE FOLLWED

1) Change directory into the beginner_tutorials package

   roscd beginner_tutorials

2) Change directory to scripts

   cd scripts

   gedit listener.py (gedit is a command used to create a text file)

Copy the below python script to listener.py (text file created in the script folder) make it executable

listener.py

```
#!/usr/bin/env python3
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # name are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '_main_':
    listener()
```

3) Make the python file executable using the command

chmod +x listener.py (This code will make the program executable)

1) Building your node

This is to make sure that the autogenerated Python code for messages and services is created.

(Go to your catkin workspace and run catkin_make)

cd ~/name of your workspace

catkin_make

2) Running a Subscriber node **(It will work only when Publishing node is Running)**

a. To add the workspace to your ROS environment you need to source the generated setup file:**(Done in new terminal)**

cd ~/ name of your workspace

source devel/setup.bash

b. Run the publishing node

rosrun beginner_tutorials listener.py

You will see something similar to

```
[INFO] [1641920814.553839]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920814.654042]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920814.753581]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920814.853538]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920814.954358]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.054191]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.153767]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.254494]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.353559]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.454451]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.553758]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.653816]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.754466]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.854366]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920815.953737]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920816.053761]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920816.154043]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920816.253636]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920816.354571]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920816.454097]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920816.553835]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920816.653539]: /listener_3395_16419208135901 heard hello world
[INFO] [1641920816.754597]: /listener_3395_16419208135901 heard hello world
```

2. Writing a Service Node and Client Node

Here we'll create the service ("add_two_ints_server") node which will receive two ints and return the sum

roscd beginner_tutorials

1) Creating the srv file.

mkdir srv

roscp rospy_tutorials AddTwoInts.srv srv/AddTwoInts.srv

Open package.xml, and make sure these two lines are added in it

```
<build_depend>message_generation</build_depend>

 <exec_depend>message_runtime</exec_depend>
```

add this in CMakeLists.txt

```
# Do not just add this line to your CMakeLists.txt, modify the existing line

find_package(catkin REQUIRED COMPONENTS

 roscpp

 rospy

 std_msgs
```

```
  message_generation
)
```

Add these commands to CMakeLists.txt

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

2) Change directory to scripts

cd scripts

gedit add_two_ints_server.py (gedit is a command used to create a text file)

add_two_ints_server.py

```
#!/usr/bin/env python3

from__future__import print_function

from beginner_tutorials.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print("Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b)))
    return AddTwoIntsResponse(req.a + req.b)
```

```
def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print("Ready to add two ints.")
    rospy.spin()

if __name__ == "_main_":
    add_two_ints_server()
```

gedit add_two_ints_client.py (gedit is a command used to create a text file)

```
#!/usr/bin/env python3

from __future__ import print_function

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
```

```
    y = int(sys.argv[2])
else:
    print(usage())
    sys.exit(1)
print("Requesting %s+%s"%(x, y))
print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

3) Make the python file executable using the command

chmod +x add_two_ints_server.py (This code will make the program executable)

chmod +x add_two_ints_client.py (This code will make the program executable)

Add this in the CMakeLists.txt

```
catkin_install_python(PROGRAMS scripts/talker.py scripts/listener.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

4) Building your node

This is to make sure that the autogenerated Python code for messages and services is created.**(Go to your catkin workspace and run catkin_make)**

cd ~/ name of your workspace

catkin_make

5) Running Service and client node

a. Run Ros master in a **new terminal 1**

roscore

b.Run the Service node**(Done in new terminal 2)**

cd ~/ name of your workspace

source devel/setup.bash

To add the workspace to your ROS environment you need to source the generated setup file

cd ~/ name of your workspace

rosrun beginner_tutorials add_two_ints_server.py

```
roshni@ThinkPad-E14:~/ran$  rosrun beginner_tutorials add_two_ints_server.py
Ready to add two ints.
```

c. Run the client node (**Open a new terminal 3)**

cd ~/ name of your workspace

source devel/setup.bash

rosrun beginner_tutorials add_two_ints_client.py 1 3

```
roshni@ThinkPad-E14:~/ran$ source devel/setup.bash
roshni@ThinkPad-E14:~/ran$ rosrun beginner_tutorials add_two_ints_client.py 1 3
Requesting 1+3
1 + 3 = 4
roshni@ThinkPad-E14:~/ran$
```

RESULT:

Experiment:2

AIM: Getting started with Turtlesim

1. Start the ROS master using the command

```
roscore
```

2. Turtlesim Node

We will start the turtlesim node and explore its properties.In a new terminal create the turtlesim node from the package turtlesim:

```
rosrun turtlesim turtlesim_node (In a new terminal)
```

You will see

[INFO] [1638162556.114153594]: Starting turtlesim with node name /turtlesim

[INFO] [1638162556.117803739]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]

The rosrun command takes the arguments [package name] [node name]. The node creates the screen image and the turtle. Here the turtle is in the center in x=5.5, y=5.5 with no rotation.

3. Keyboard Control

In a third window, we execute a node that allows keyboard control of the turtle. Roscore is running in one window and turtlesim_node in another.

```
rosrun turtlesim turtle_teleop_key(In a new terminal)
```

You will see

Reading from keyboard

- - - - - - - - - - - - - - - - - -

Use arrow keys to move the turtle.

Up arrow Turtle In Turtle's x direction

Down arrow Turtle In Turtles's -x direction

Right arrow Rotate CW

Left arrow Rotate CCW

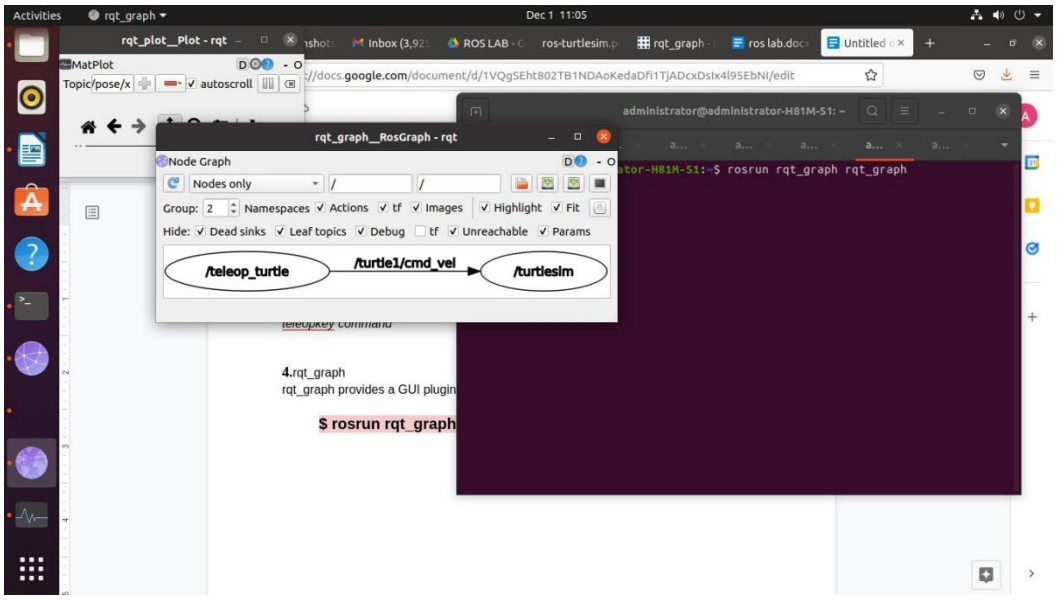
Pressing the arrow keys on the keyboard should cause the turtle to move around the screen. Note that to move the turtle you must have the cursor in the terminal with teleopkey command.

4.rqt_graph

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

rosrun rqt_graph rqt_graph(In a new terminal)

You will see



5.ROS Nodes with Turtlesim

A node really is an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.

5a . rosnode

rosnode displays information about the ROS nodes that are currently running. It is a command-line tool for printing information about ROS Nodes.

Commands:

rosnode ping test connectivity to node
rosnode list list active nodes
rosnode info print information about node
rosnode machine list nodes running on a particular machine or list machines
rosnode kill kill a running node
rosnode cleanup purge registration information of unreachable nodes

5b. rosnode list

The rosnode list command lists the active nodes:

You will see
/rosout
/teleop_turtle
/turtlesim

5c.rosnode info /turtlesim

The rosnode info command returns information about a specific node.

You will see
Node [/turtlesim]
Publications: (This information is sent to nodes listening to /turtlesim)
* /rosout [rosgraph_msgs/Log]
* /turtle1/color_sensor [turtlesim/Color]

* /turtle1/pose [turtlesim/Pose]

Subscriptions:(This node will listen for command velocities)

* /turtle1/cmd_vel [geometry_msgs/Twist]

Services:(We can use ROS services to manipulate the turtle and perform other operations.)

* /clear
* /kill
* /reset (The format is $rosservice call <service> <arguments>)
* /spawn
* /turtle1/set_pen
* /turtle1/teleport_absolute
* /turtle1/teleport_relative
* /turtlesim/get_loggers
* /turtlesim/set_logger_level

contacting node http://administrator-H81M-S1:43435/ ...

Pid: 5989

Connections:

* topic: /rosout
    * to: /rosout
    * direction: outbound (42467 - 127.0.0.1:49876) [27]
    * transport: TCPROS
* topic: /turtle1/pose
    * to: /rqt_gui_py_node_6948
    * direction: outbound (42467 - 127.0.0.1:50250) [29]
    * transport: TCPROS
* topic: /turtle1/cmd_vel
    * to: /teleop_turtle (http://administrator-H81M-S1:43945/)
    * direction: inbound (50112 - administrator-H81M-S1:39295) [30]
    * transport: TCPROS

The node /turtlesim publishes three topics and subscribes to the /turtle1/cmd_vel topic. The services for the node are also listed

6.ROS Topics

The turtlesim_node and the turtle_teleop_key node are communicating with each other over a ROS Topic. turtle_teleop_key is publishing the key strokes on a topic, while turtlesim subscribes to the same topic to receive the key strokes.

6a.rostopic -h

The rostopic tool allows you to get information about ROS topics.
It will give the available sub-commands for rostopic

You will see
rostopic bw display bandwidth used by topic
rostopic echo print messages to screen
rostopic hz display publishing rate of topic
rostopic list print information about active topics
rostopic pub publish data to topic
rostopic type print topic type

6b.rostopic echo /turtle1/cmd_vel(In a new terminal)

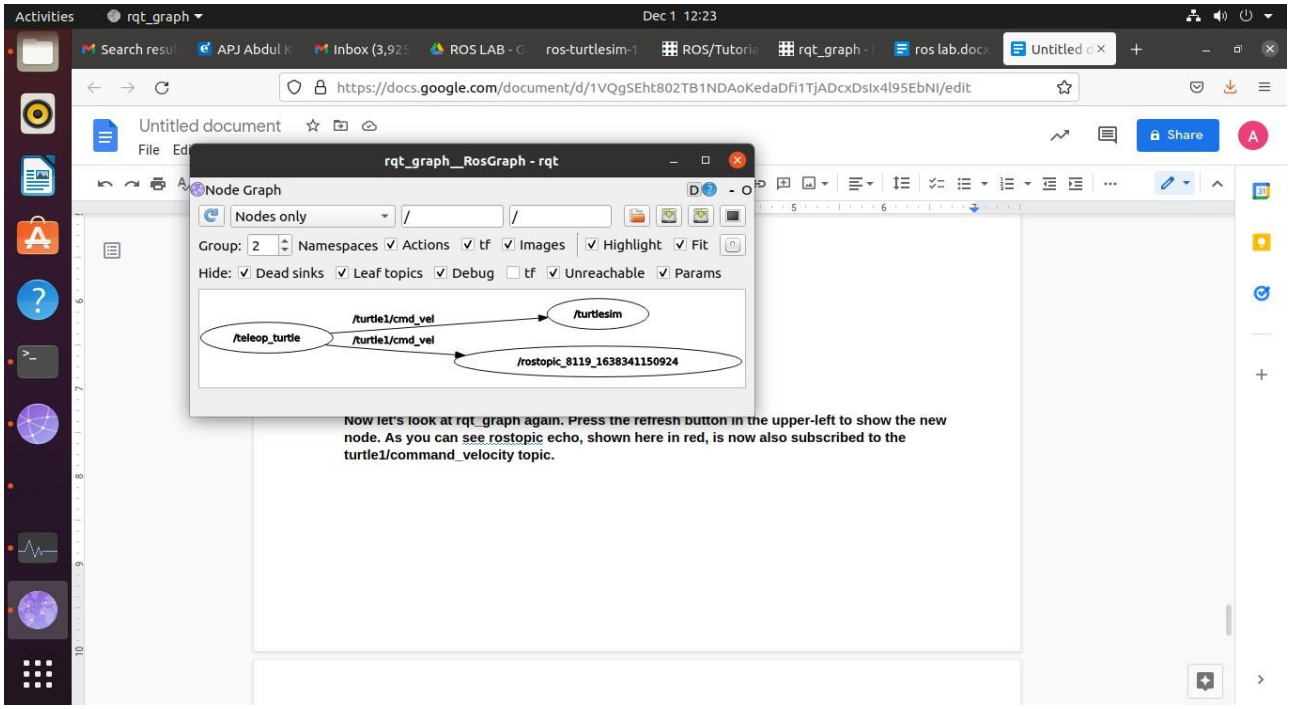
rostopic echo shows the data published on a topic. Let's look at the command velocity data published by the turtle_teleop_key node.

You will see
linear:
x: -2.0
y: 0.0

```
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: -2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: -2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: -2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

```
linear:
  x: -2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

Now let's look at rqt_graph again. Press the refresh button in the upper-left to show the new node. As you can see rostopic echo, shown here in red, is now also subscribed to the turtle1/command_velocity topic.



rostopic list returns a list of all topics currently subscribed to and published.

6c.rostopic list -h(In a new terminal)

You will see

Usage: rostopic list [/namespace]

Options:

-h, --help show this help message and exit

-b BAGFILE, --bag=BAGFILE list topics in .bag file

-v, --verbose list full details about each topic

-p list only publishers

-s list only subscribers

--host group by host name

6d.rostopic list -v

This displays a verbose list of topics to publish to and subscribe to and their type.

You will see

Published topics:

* /rosout_agg [rosgraph_msgs/Log] 1 publisher
* /rosout [rosgraph_msgs/Log] 5 publishers
* /turtle1/pose [turtlesim/Pose] 1 publisher
* /turtle1/color_sensor [turtlesim/Color] 1 publisher
* /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher

Subscribed topics:

* /rosout [rosgraph_msgs/Log] 1 subscriber
* /turtle1/cmd_vel [geometry_msgs/Twist] 2 subscribers
* /turtle1/pose [turtlesim/Pose] 1 subscriber
* /statistics [rosgraph_msgs/TopicStatistics] 1 subscriber

7. ROS Messages

Communication on topics happens by sending ROS messages between nodes. For the publisher (turtle_teleop_key) and subscriber (turtlesim_node) to communicate, the publisher and subscriber must send and receive the same type of message. This means that a topic type is defined by the message type published on it. The type of the message sent on a topic can be determined using rostopic type.

rostopic type [topic]

7a.rostopic type /turtle1/cmd_vel

rostopic type returns the message type of any topic being published.

You will see

geometry_msgs/Twist

We can look at the details of the message using rosmsg:

7b.rosmsg show geometry_msgs/Twist

You will see

geometry_msgs/Vector3 linear

float64 x

float64 y

float64 z

geometry_msgs/Vector3 angular

float64 x

float64 y

float64 z

8. Let's use rostopic with messages.

Using rostopic pub

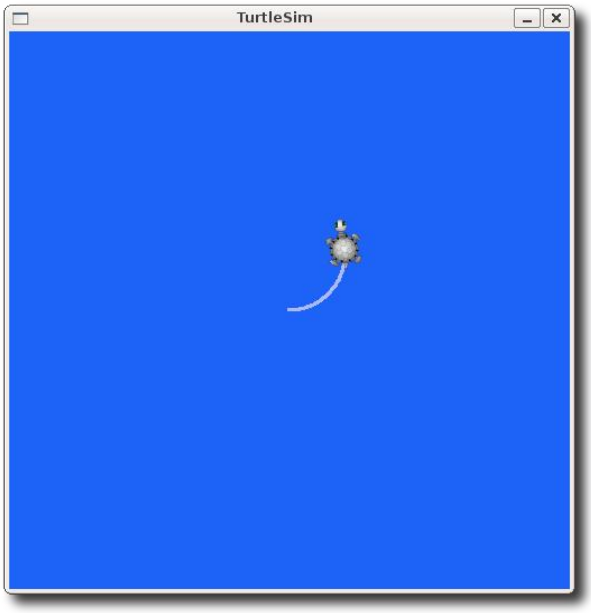
rostopic pub publishes data on to a topic currently advertised.

Usage:rostopic pub [topic] [msg_type] [args]

8a.rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'

This command will send a single message to turtlesim telling it to move with a linear velocity of 2.0, and an angular velocity of 1.8 .

You will see

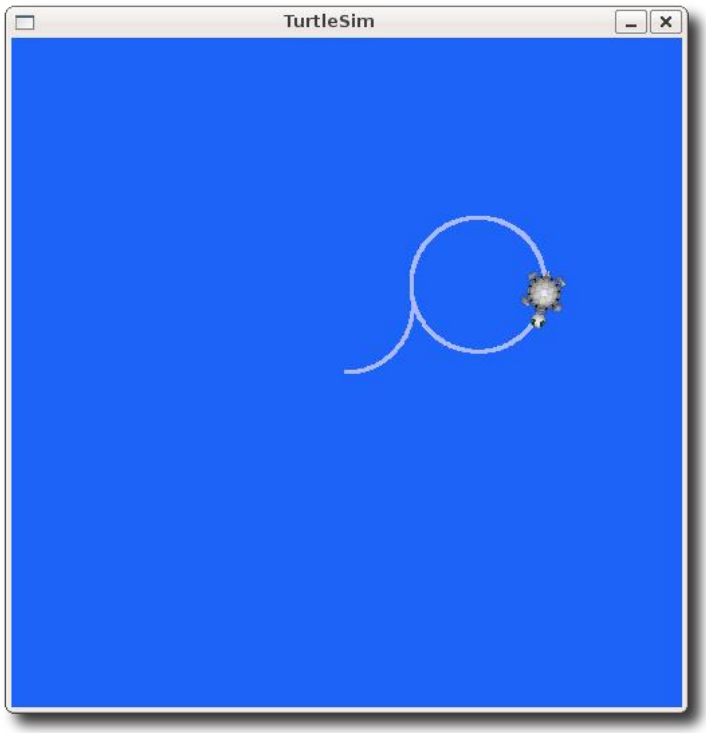


This command will publish messages to a given topic: This option (dash-one) ()causes rostopic to only publish one message then exit: This is the name of the topic to publish to.This option (double-dash) tells the option parser that none of the following arguments is an option. This is required in cases where your arguments have a leading dash -, like negative numbers.

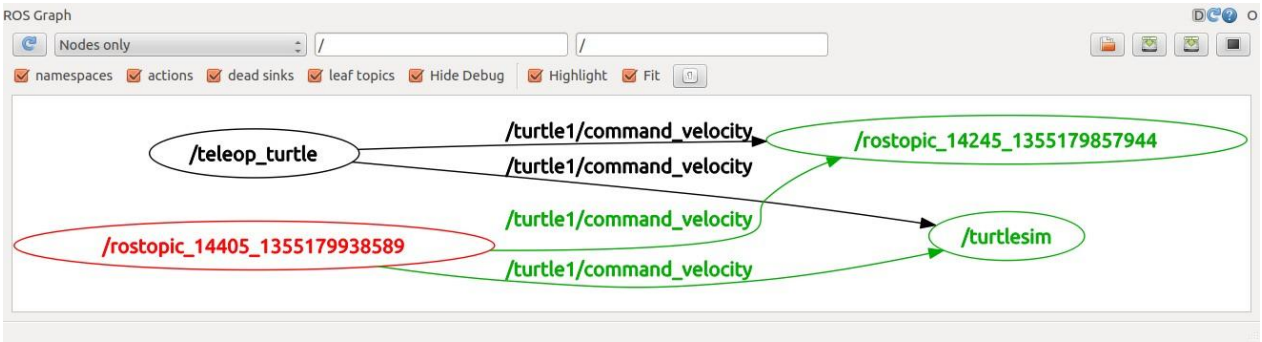
You may have noticed that the turtle has stopped moving; this is because the turtle requires a steady stream of commands at 1 Hz to keep moving. We can publish a steady stream of commands using rostopic pub -r command:

8b.rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'

This publishes the velocity commands at a rate of 1 Hz on the velocity topic.
You will see



We can also look at what is happening in rqt_graph. Press the refresh button in the upper-left. The rostopic pub node (here in red) is communicating with the rostopic echo node (here in green):



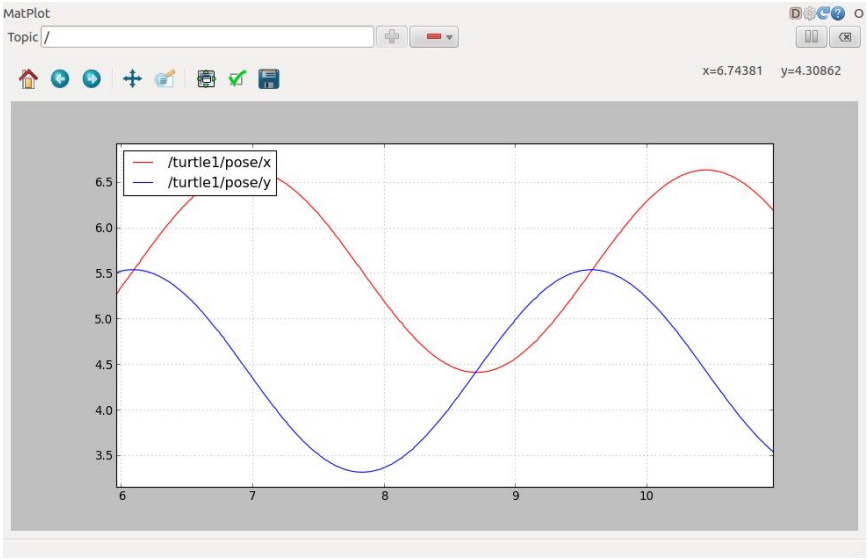
As you can see the turtle is running in a continuous circle.

9.rqt_plot

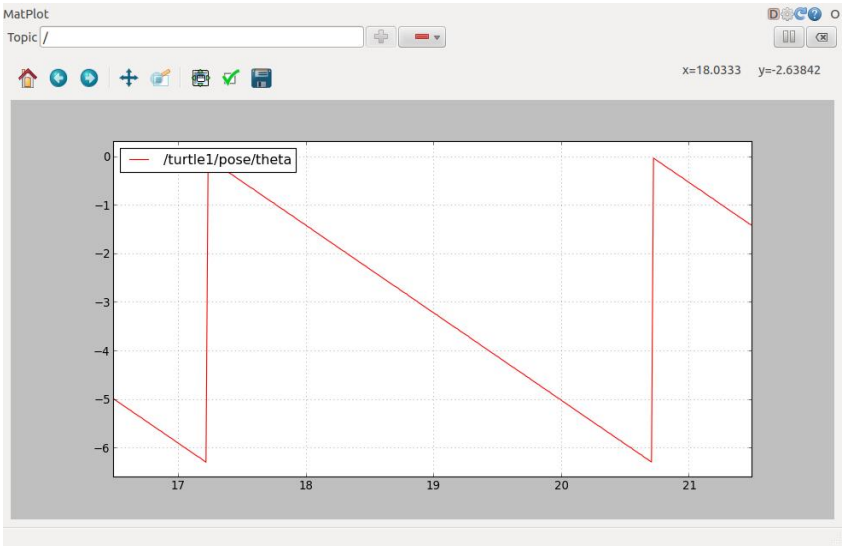
rqt_plot displays a scrolling time plot of the data published on topics. Here we'll use rqt_plot to plot the data being published on the /turtle1/pose topic. First, start rqt_plot by typing

rosrun rqt_plot rqt_plot(In a new terminal.)

In the new window that should pop up, a text box in the upper left corner gives you the ability to add any topic to the plot. Typing /turtle1/pose/x will highlight the plus button, previously disabled. Press it and repeat the same procedure with the topic /turtle1/pose/y. You will now see the turtle's x-y location plotted in the graph.



Pressing the minus button shows a menu that allows you to hide the specified topic from the plot. Hiding both the topics you just added and adding /turtle1/pose/theta will result in the plot shown in the next figure.



10. ROS Services

Services are another way that nodes can communicate with each other. Services allow nodes to send a request and receive a response.

Usage:

rosservice list print information about active services
rosservice call call the service with the provided args
rosservice type print service type
rosservice find find services by service type
rosservice uri print service ROSRPC uri

10a. rosservice list

The list command shows us that the turtlesim node provides nine services: reset, clear, spawn, kill, turtle1/set_pen, /turtle1/teleport_absolute, /turtle1/teleport_relative, turtlesim/get_loggers, and turtlesim/set_logger_level. There are also two services related to the separate rosout node: /rosout/get_loggers and /rosout/set_logger_level.

You will see

/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative

/turtlesim/get_loggers
/turtlesim/set_logger_l

Usage:
rosservice type [service]

10b. rosservice type /clear

We can find out what type the clear service is:

You will see
std_srvs/Empty

Usage:
rosservice call [service] [args]

10c.rosservice call /clear
it clears the background of the turtlesim_node.

10d.rosservice call reset
Resets the turtlesim to the start configuration and sets the background color to the value of the background.

10f.rosservice call /spawn 2 2 0.2 turtle2
This service lets us spawn a new turtle at a given location and orientation. The name field is optional.The service call returns with the name of the newly created turtle

- name: turtle2

10g.rosservice call /turtle1/set_pen 255 0 255 3 on
Sets the pen's color (r g b), width (width), and turns the pen on and off (off).

10h.rosservice call /turtle1/teleport_absolute 1 1 0

The turtle can be moved using the rosservice teleport option. The format of the position is [x y theta].

10i.rosservice call /turtle1/teleport_relative 1 0

The relative teleport option moves the turtle with respect to its present position. The arguments are [linear, angle]

11. ROS Parameters

Using rosparam allows you to store and manipulate data on the ROS Parameter Server. rosparam has many commands that can be used on parameters, as shown below:

Usage:

rosparam set set parameter
rosparam get get parameter
rosparam load load parameters from file
rosparam dump dump parameters to file
rosparam delete delete parameter
rosparam list list parameter names

11a.rosparam list

what parameters are currently on the param server:

/rosdistro
/roslaunch/uris/host_nxt__43407
/rosversion
/run_id
/turtlesim/background_b
/turtlesim/background_g
/turtlesim/background_r

11b.rosparam set /turtlesim/background_r 150

This command will change the red channel of the background color:

This changes the parameter value, now we have to call the clear service for the parameter change to take effect:

$ rosservice call /clear

11c.rosparam get /turtlesim/background_g

This command will get the value of the green background channel:

We can also use rosparam get / to show us the contents of the entire Parameter Server.

11d.rosparam get /

```
rosdistro: 'noetic’
roslaunch:
  uris:
    host_nxt_43407: http://nxt:43407/
rosversion: '1.15.5


  '
run_id: 7ef687d8-9ab7-11ea-b692-fcaa1494dbf9
turtlesim:
  background_b: 255
  background_g: 86
  background_r: 69
```

Recording and playing back data

Recording data (creating a bag file)

This section will instruct you how to record topic data from a running ROS system. The topic data will be accumulated in a bag file.

First, execute the following commands in separate terminals:

1.roscore (Terminal 1)

2. rosrun turtlesim turtlesim_node (Terminal 2)

3. rosrun turtlesim turtle_teleop_key (Terminal 3)

First lets examine the full list of topics that are currently being published in the running system. To do this, open a new terminal and execute the command:

4. rostopic list -v (Terminal 4)

You will see

Published topics:

* /turtle1/color_sensor [turtlesim/Color] 1 publisher
* /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
* /rosout [rosgraph_msgs/Log] 2 publishers
* /rosout_agg [rosgraph_msgs/Log] 1 publisher
* /turtle1/pose [turtlesim/Pose] 1 publisher

Subscribed topics:

* /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
* /rosout [rosgraph_msgs/Log] 1 subscriber

We now will record the published data. Open a new terminal window. In this window run the following commands:

5. mkdir bagfiles (Terminal 5)
6. cd bagfiles
7. rosbag record -a

Here we are just making a temporary directory to record data and then running rosbag record with the option -a, indicating that all published topics should be accumulated in a bag file.

Move back to the terminal window with turtle_teleop and move the turtle around for 10 or so seconds.

In the window running rosbag record exit with a Ctrl-C.

Now examine the contents of the directory ~/bagfiles. You should see a file with a name that begins with the year, date, and time and the suffix .bag.

This is the bag file that contains all topics published by any node in the time that rosbag record was running.

Examining and playing the bag file

We've recorded a bag file using rosbag record .Now we can examine it and play it back using the commands rosbag info and rosbag play. First we are going to see what's recorded in the bag file. We can do the info command -- this command checks the contents of the bag file without playing it back. Execute the following command from the bagfiles directory:

8.rosbag info <name of your bagfile> (Terminal 5)

You will see

path: 2021-12-04-11-39-45.bag
version: 2.0
duration: 8:33s (513s)
start: Dec 04 2021 11:39:45.48 (1638598185.48)
end: Dec 04 2021 11:48:19.14 (1638598699.14)
size: 4.3 MB
messages: 64106
compression: none [5/5 chunks]
types: geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
rosgraph_msgs/Log [acffd30cd6b6de30f120938c17c593fb]
turtlesim/Color [353891e354491c51aabe32df673fb446]
turtlesim/Pose [863b248d5016ca62ea2e895ae5265cf9]
topics: /rosout 3 msgs : rosgraph_msgs/Log
/turtle1/cmd_vel 160 msgs : geometry_msgs/Twist
/turtle1/color_sensor 31971 msgs : turtlesim/Color
/turtle1/pose 31972 msgs : turtlesim/Pose

This tells us topic names and types as well as the number (count) of each message topic contained in the bag file.

The next step is to replay the bag file to reproduce behavior in the running system.

First kill the teleop program that may be still running from the previous section - a Ctrl-C in the terminal where you started turtle_teleop_key.

Leave turtlesim running. In a terminal window run the following command in the directory where you took the original bag file:

9.rosservice call reset

10.rosbag play <your bagfile>

In this window you should immediately see something like:

[INFO] [1418271315.162885976]: Opening 2014-12-10-20-08-34.bag

Waiting 0.2 seconds after advertising topics... done.

In its default mode rosbag play will wait for a certain period (.2 seconds) after advertising each message before it actually begins publishing the contents of the bag file.

****

10e.rosservice call kill turtle name

Kills a turtle by name.

RESULT:

EXPERIMENT:3

AIM: Familiarisation with Rviz -- Markers: Sending Basic Shapes -- use visualization_msgs/Marker messages to send basic shapes, Interactive Markers: Writing a Simple Interactive Marker Server, Basic Controls

Familiarisation with Rviz

Rviz, abbreviation for **ROS visualization**, is a powerful 3D visualization tool for ROS. It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information.

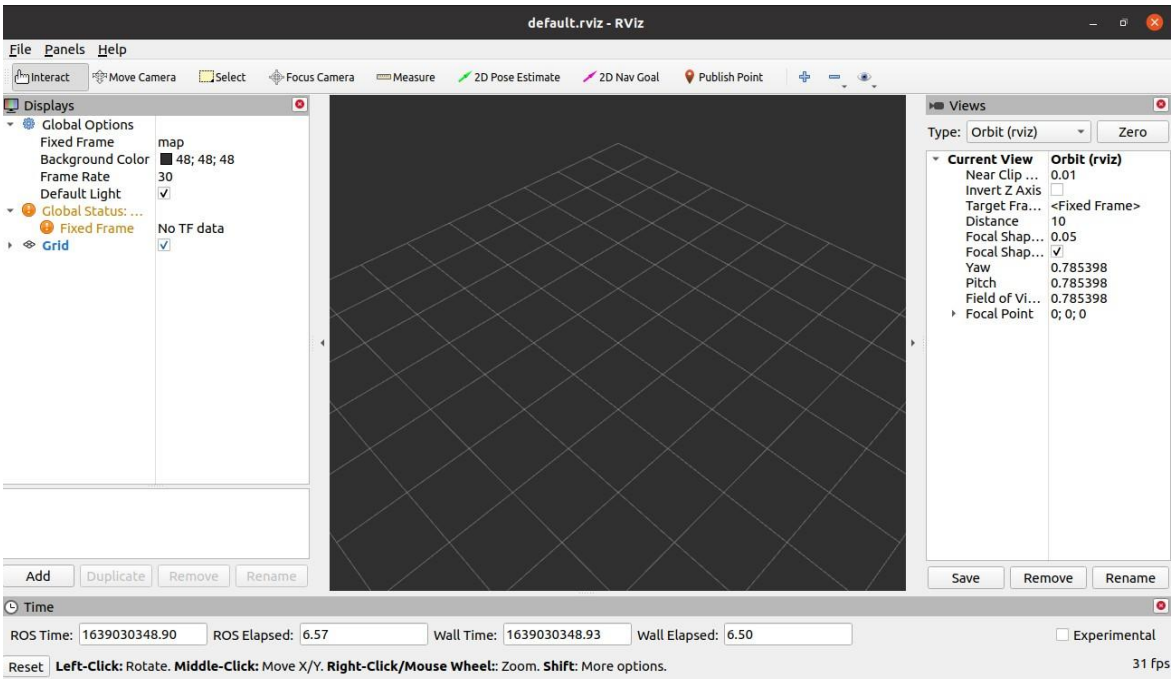
Run Ros master in a **new terminal 1**

roscore

Start the visualizer

rosrun rviz rviz (terminal 2)

When rviz starts for the first time, you will see an empty window:



The big black thing is the 3D view (empty because there is nothing to see). On the left is the Displays list, which will show any displays you have loaded.

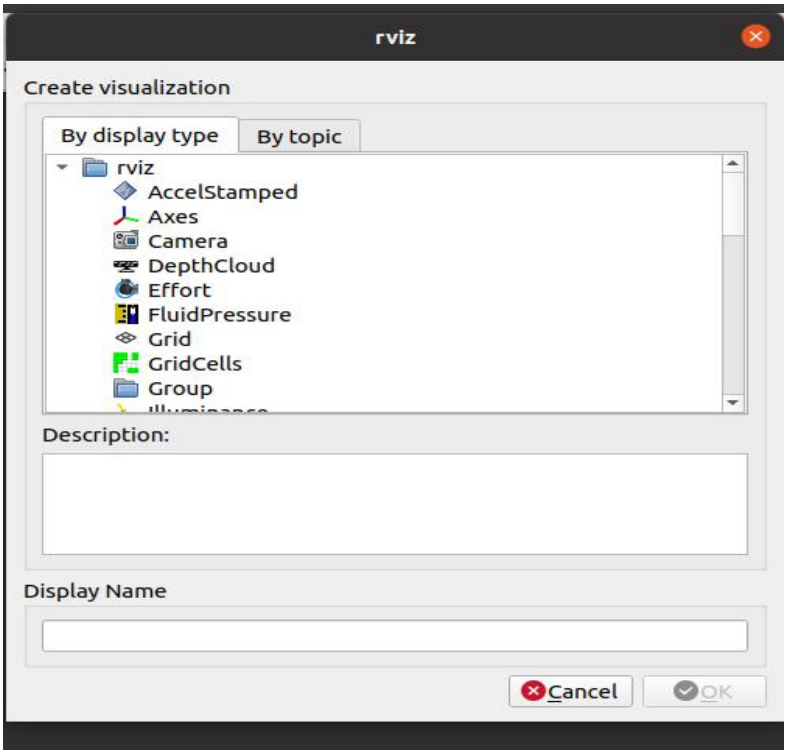
Displays

A display is something that draws something in the 3D world

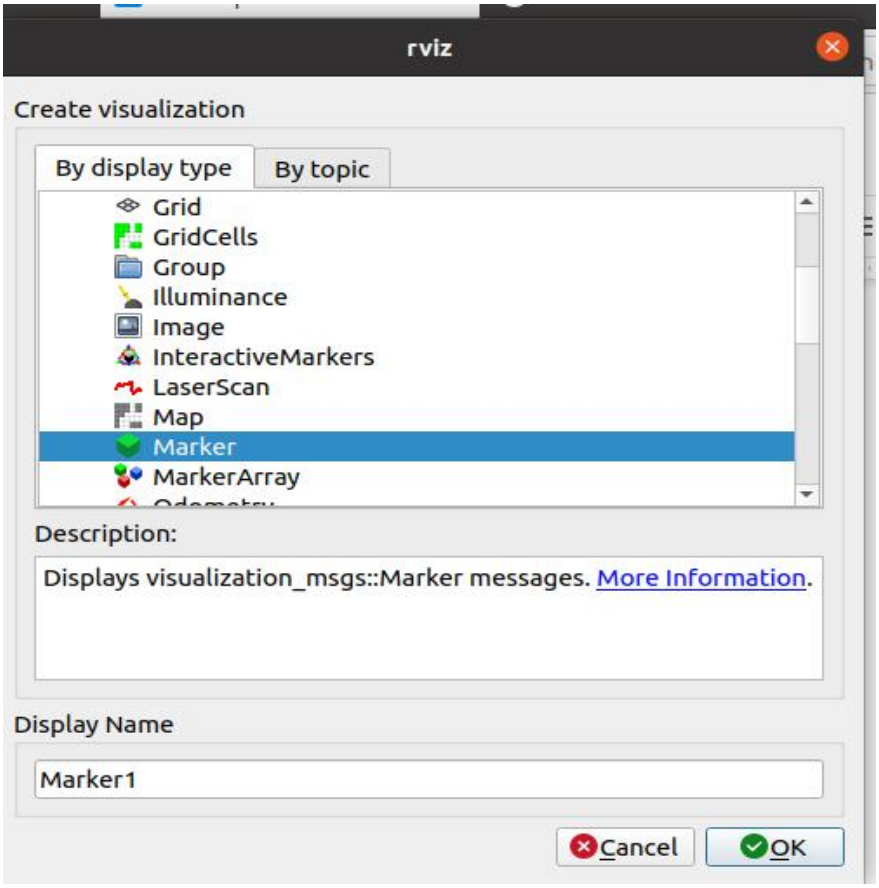
Adding a new display

To add a display, click the Add button at the bottom:

This will pop up the new display dialog:



The list at the top contains the display *type*. The type details what kind of data this display will visualize. The text box in the middle gives a description of the selected display type. Finally, you must give the display a unique name.



Markers: Sending Basic Shapes

The Marker Display lets you visualize data in rviz without rviz knowing anything about interpreting that data. Instead, primitive objects are sent to the display through visualization_msgs/Marker messages, which let you show things like arrows, boxes, spheres and lines.

cd name of your workspace

cd src

STEPS TO BE FOLLWED

1. Create a package called using_markers

   catkin_create_pkg using_markers roscpp visualization_msgs

2. Change directory into src folder in the using_markers package

   cd using_markers

cd src

2. Paste the following into basic_shapes.cpp

gedit basic_shapes.cpp

basic_shapes.cpp

```
#include <ros/ros.h>
#include <visualization_msgs/Marker.h>

int main( int argc, char** argv )
{
  ros::init(argc, argv, "basic_shapes");//name of the node
  ros::NodeHandle n;//NodeHandle is the main access point to communications with the //ROS
system.

  ros::Rate r(1);//ROS Rate at 1Hz
  ros::Publisher marker_pub = n.advertise<visualization_msgs::Marker>("visualization_marker",
1);//advertise() //function is how you tell ROS that you want to publish on a given topic name


  // Set our initial shape type to be a cube
  uint32_t shape = visualization_msgs::Marker::CUBE;

  while (ros::ok())
  {
    visualization_msgs::Marker marker;
    // Set the frame ID and timestamp. See the TF tutorials for information on these.
    marker.header.frame_id = "my_frame";
    marker.header.stamp = ros::Time::now();

    // Set the namespace and id for this marker.  This serves to create a unique ID
    // Any marker sent with the same namespace and id will overwrite the old one
    marker.ns = "basic_shapes";
    marker.id = 0;

    // Set the marker type. Initially this is CUBE, and cycles between that and SPHERE, ARROW,
and CYLINDER
    marker.type = shape;

    // Set the marker action. Options are ADD, DELETE, and new in ROS Indigo: 3
(DELETEALL)
    marker.action = visualization_msgs::Marker::ADD;

    // Set the pose of the marker. This is a full 6DOF pose relative to the frame/time specified in
the header
    marker.pose.position.x = 0;
    marker.pose.position.y = 0;
    marker.pose.position.z = 0;
    marker.pose.orientation.x = 0.0;
```

```
  marker.pose.orientation.y = 0.0;
  marker.pose.orientation.z = 0.0;
  marker.pose.orientation.w = 1.0;

  // Set the scale of the marker -- 1x1x1 here means 1m on a side
  marker.scale.x = 1.0;
  marker.scale.y = 1.0;
  marker.scale.z = 1.0;

  // Set the color -- be sure to set alpha to something non-zero!
  marker.color.r = 0.0f;
  marker.color.g = 1.0f;
  marker.color.b = 0.0f;
  marker.color.a = 1.0;

  marker.lifetime = ros::Duration();

  // Publish the marker
  while (marker_pub.getNumSubscribers() < 1)
  {
    if (!ros::ok())
    {
      return 0;
    }
    ROS_WARN_ONCE("Please create a subscriber to the marker");
    sleep(1);
  }
  marker_pub.publish(marker);

  // Cycle between different shapes
  switch (shape)
  {
  case visualization_msgs::Marker::CUBE:
    shape = visualization_msgs::Marker::SPHERE;
    break;
  case visualization_msgs::Marker::SPHERE:
    shape = visualization_msgs::Marker::ARROW;
    break;
  case visualization_msgs::Marker::ARROW:
    shape = visualization_msgs::Marker::CYLINDER;
    break;
  case visualization_msgs::Marker::CYLINDER:
    shape = visualization_msgs::Marker::CUBE;
    break;
  }

  r.sleep();
 }
}
```

add this in CMakeLists.txt

```
add_executable(basic_shapes src/basic_shapes.cpp)
target_link_libraries(basic_shapes ${catkin_LIBRARIES})
```

3.Building your node

(Go to your catkin workspace and run catkin_make)

cd ~/ name of your workspace

catkin_make

4.Running the code

a. Run Ros master in a **new terminal 1**

roscore

a.Run the basic_shapes node**(Done in new terminal 2)**

cd ~/ name of your workspace

source devel/setup.bash

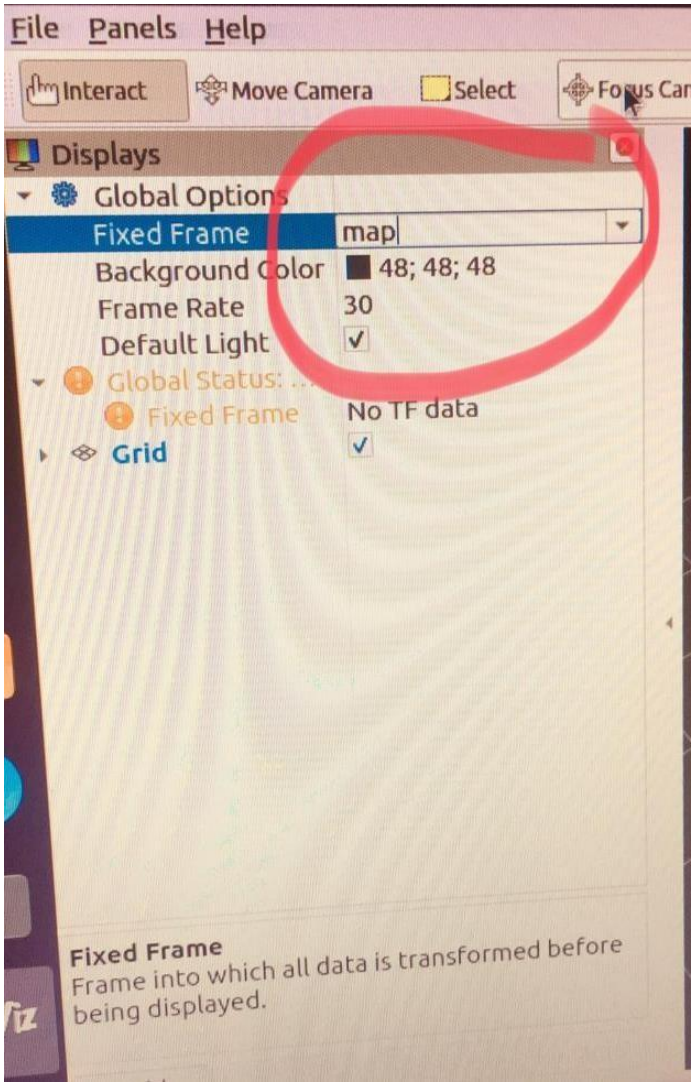
rosrun using_markers basic_shapes

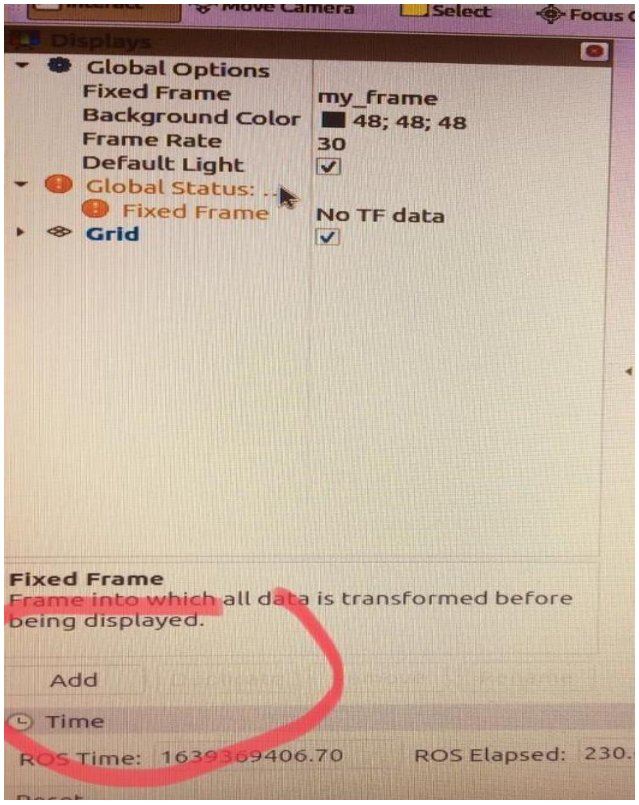
```
administrator@administrator-B365M-GAMING-HD:~/rviz_ws$ rosrun using_markers basi
c_shapes
[ WARN] [1639368847.878098517]: Please create a subscriber to the marker
```

5. Viewing the Markers

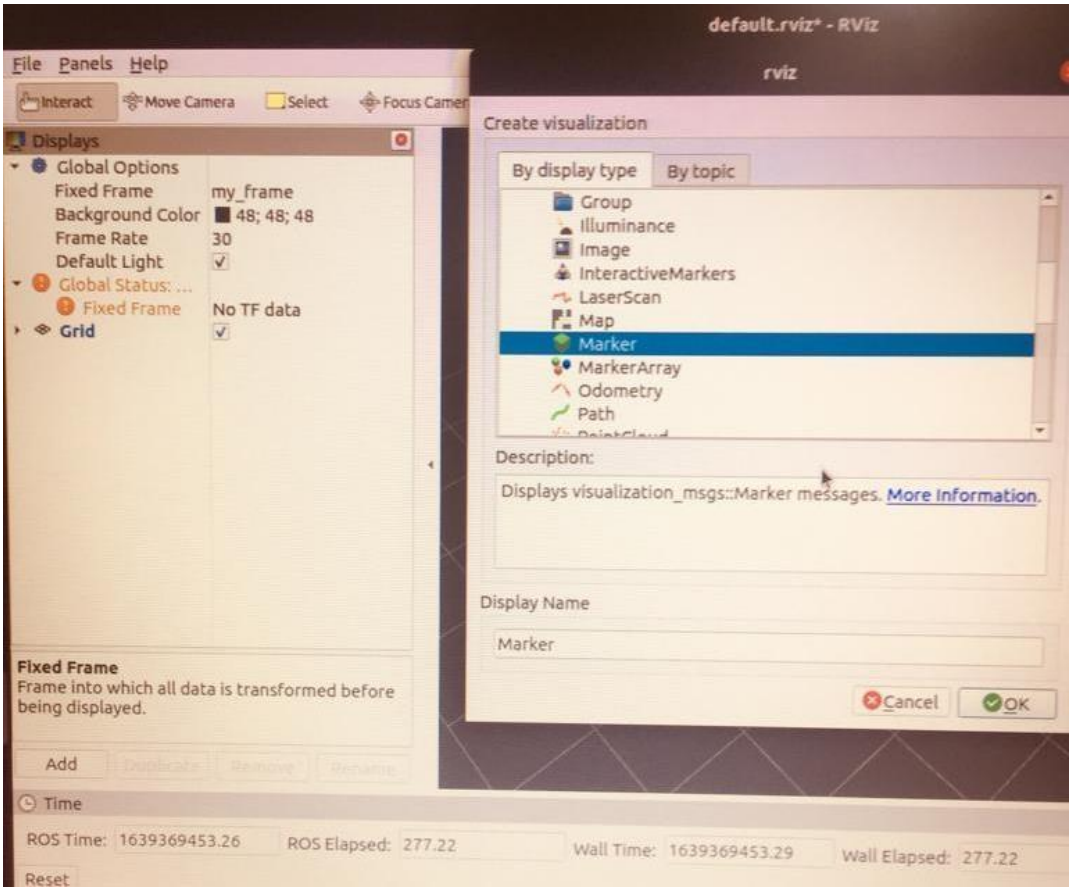
rosrun rviz rviz (new terminal 2)

In the rviz , set the Fixed Frame field to my_frame

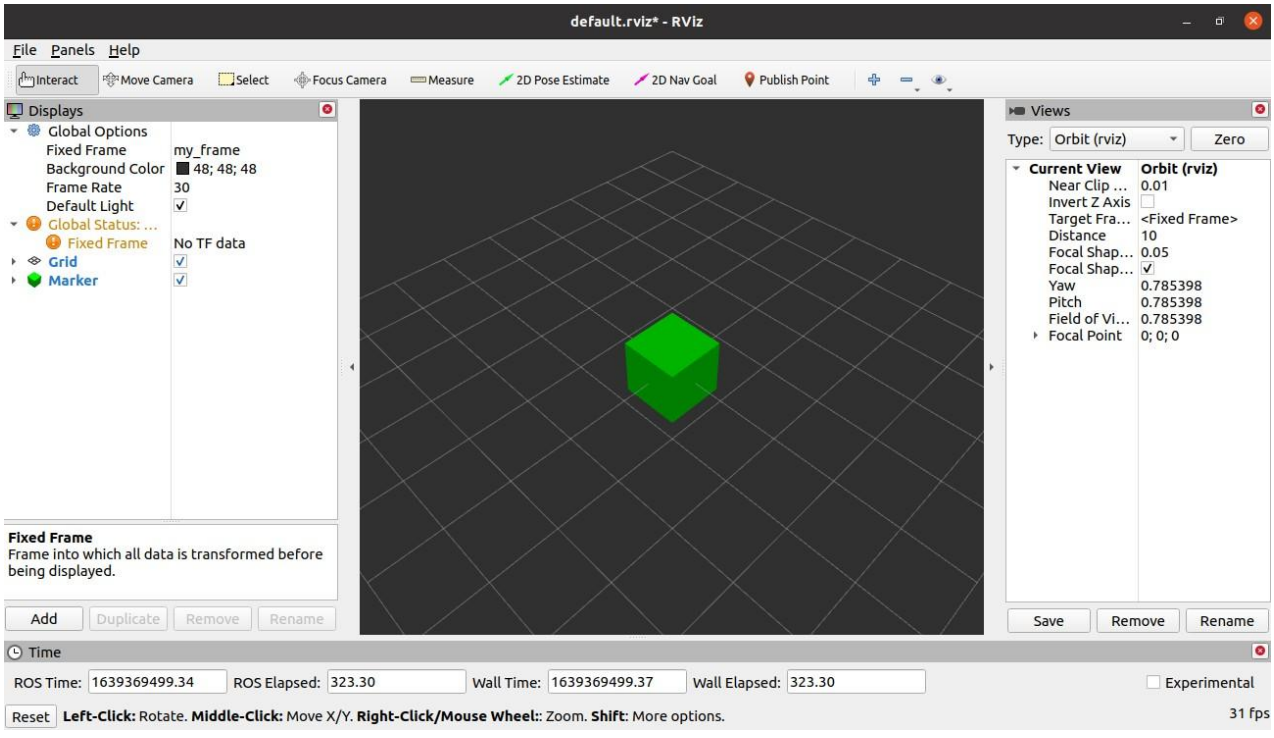

File Panels Help
Interact
Move Camera
Select
Displays
Global Options
Fixed Frame
map
Background Color
48; 48; 48
Frame Rate
30
Default Light
Global Status:
Fixed Frame
No TF data
Grid
Fixed Frame
Frame into which all data is transformed before being displayed.



add a Marker display


default.rviz* - RViz
File Panels Help
Interact
Move Camera
Select
Focus Camer
rviz
Displays
Global Options
Fixed Frame
my_frame
Background Color
48; 48; 48
Frame Rate
30
Default Light
Global Status: ...
Fixed Frame
No TF data
Grid
Create visualization
By display type
By topic
Group
Illuminance
Image
InteractiveMarkers
LaserScan
Map
Marker
MarkerArray
Odometry
Path
Description:
Displays visualization_msgs::Marker messages. More Information.
Display Name
Marker
Cancel
OK
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Time
ROS Time: 1639369453.26
ROS Elapsed: 277.22
Wall Time: 1639369453.29
Wall Elapsed: 277.22
Reset

Output:


default.rviz* - RViz
File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
my_frame
Background Color
48; 48; 48
Frame Rate
30
Default Light
Global Status: ...
Fixed Frame
No TF data
Grid
Marker
Views
Type: Orbit (rviz)
Zero
Current View
Orbit (rviz)
Near Clip ...
0.01
Invert Z Axis
Target Fra...
<Fixed Frame>
Distance
10
Focal Shap...
0.05
Yaw
0.785398
Pitch
0.785398
Field of Vi...
0.785398
Focal Point
0; 0; 0
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Duplicate
Remove
Rename
Save
Time
ROS Time: 1639369499.34
ROS Elapsed: 323.30
Wall Time: 1639369499.37
Wall Elapsed: 323.30
Experimental
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
31 fps

Interactive Markers: Writing a Simple Interactive Marker Server

Interactive markers are similar to the "regular" markers, however they allow the user to interact with them by changing their position or rotation, clicking on them or selecting something from a context menu assigned to each marker.

If you want to create a node providing a set of interactive markers, you need to instantiate an InteractiveMarkerServer object. This will handle the connection to the client (usually RViz) and make sure that all changes you make are being transmitted and that your application is being notified of all the actions the user performs on the interactive markers.

1. Change directory into src folder in the using_markers package

   roscd using_markers

   cd src

   gedit simple_marker.cpp

2. Paste the following into simple_marker.cpp

```
#include <ros/ros.h>

#include <interactive_markers/interactive_marker_server.h>
//function processFeedback handles feedback messages from RViz by printing out the //position.
void processFeedback(
    const visualization_msgs::InteractiveMarkerFeedbackConstPtr &feedback )
{
  ROS_INFO_STREAM( feedback->marker_name << " is now at "
      << feedback->pose.position.x << ", " << feedback->pose.position.y
      << ", " << feedback->pose.position.z );
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "simple_marker");

  // create an interactive marker server on the topic namespace simple_marker
  interactive_markers::InteractiveMarkerServer server("simple_marker");

  // create an interactive marker for our server
  visualization_msgs::InteractiveMarker int_marker;
  int_marker.header.frame_id = "base_link";
```

```
  int_marker.header.stamp=ros::Time::now();
  int_marker.name = "my_marker";
  int_marker.description = "Simple 1-DOF Control";

  // create a grey box marker
  visualization_msgs::Marker box_marker;
  box_marker.type = visualization_msgs::Marker::CUBE;
  box_marker.scale.x = 1;
  box_marker.scale.y = 1;
  box_marker.scale.z = 1;
  box_marker.color.r = 0.5;
  box_marker.color.g = 0.5;
  box_marker.color.b = 0.5;
  box_marker.color.a = 1.0;

  // create a non-interactive control which contains the box
  visualization_msgs::InteractiveMarkerControl box_control;
  box_control.always_visible = true;
  box_control.markers.push_back( box_marker );

  // add the control to the interactive marker
  int_marker.controls.push_back( box_control );

  // create a control which will move the box
  // this control does not contain any markers,
  // which will cause RViz to insert two arrows
  visualization_msgs::InteractiveMarkerControl rotate_control;
  rotate_control.name = "move_x";
  rotate_control.interaction_mode =
      visualization_msgs::InteractiveMarkerControl::MOVE_AXIS;

  // add the control to the interactive marker
  int_marker.controls.push_back(rotate_control);

  // add the interactive marker to our collection &
  // tell the server to call processFeedback() when feedback arrives for it
  server.insert(int_marker, &processFeedback);

  // 'commit' changes and send to all clients
  server.applyChanges();

  // start the ROS main loop
  ros::spin();
}
```

Replace this in CMakeLists.txt

```
add_executable(simple_marker src/simple_marker.cpp)
target_link_libraries(simple_marker ${catkin_LIBRARIES})
```

Replace find package function with

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  visualization_msgs
  interactive_markers
)
```

cd ~/ name of your workspace

catkin_make

source devel/setup.bash

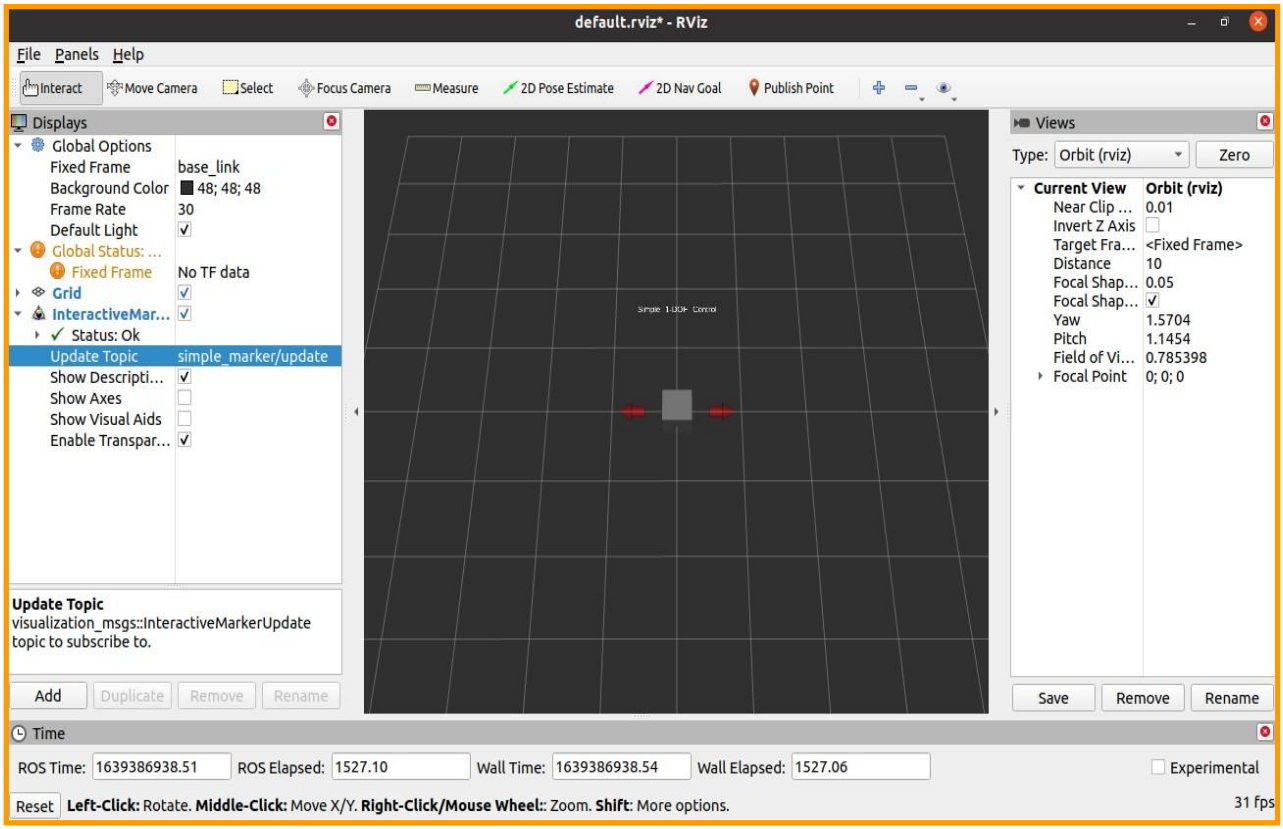
rosrun using_markers simple_marker

rosrun rviz rviz(new terminal)

In the rviz , set the Fixed Frame field to base_link

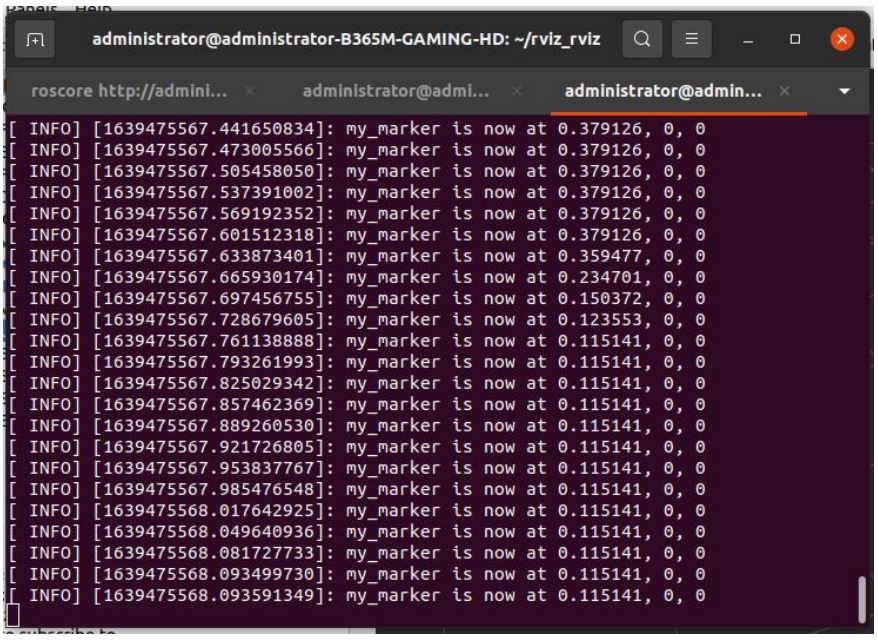
Add Interactive markers to display

In interactive markers, set update Topic to simple_marker/update



The marker shown in rviz is a 1 DOF interactive marker. To move the marker, place the mouse pointer in the arrow shown along with the marker and drag it in both directions.

In the terminal you will see the output like this



Interactive Markers: Basic Controls

1. Change directory into src folder in the using_markers package

   roscd using_markers

   cd src

   gedit basic_controls.cpp

2. Paste the following into basic_controls.cpp

```
#include <ros/ros.h>

#include <interactive_markers/interactive_marker_server.h>
#include <interactive_markers/menu_handler.h>

#include <tf/transform_broadcaster.h>
#include <tf/tf.h>

#include <math.h>

using namespace visualization_msgs;


// %Tag(vars)%
boost::shared_ptr<interactive_markers::InteractiveMarkerServer> server;
interactive_markers::MenuHandler menu_handler;
// %EndTag(vars)%


// %Tag(Box)%
Marker makeBox( InteractiveMarker &msg )
{
 Marker marker;

 marker.type = Marker::CUBE;
 marker.scale.x = msg.scale * 0.45;
 marker.scale.y = msg.scale * 0.45;
 marker.scale.z = msg.scale * 0.45;
 marker.color.r = 0.5;
 marker.color.g = 0.5;
 marker.color.b = 0.5;
 marker.color.a = 1.0;

 return marker;
}
```

```
InteractiveMarkerControl& makeBoxControl( InteractiveMarker &msg )
{
 InteractiveMarkerControl control;
 control.always_visible = true;
 control.markers.push_back( makeBox(msg) );
 msg.controls.push_back( control );

 return msg.controls.back();
}
// %EndTag(Box)%

// %Tag(frameCallback)%
void frameCallback(const ros::TimerEvent&)
{
 static uint32_t counter = 0;

 static tf::TransformBroadcaster br;

 tf::Transform t;

 ros::Time time = ros::Time::now();

 t.setOrigin(tf::Vector3(0.0, 0.0, sin(float(counter)/140.0) * 2.0));
 t.setRotation(tf::Quaternion(0.0, 0.0, 0.0, 1.0));
 br.sendTransform(tf::StampedTransform(t, time, "base_link", "moving_frame"));

 t.setOrigin(tf::Vector3(0.0, 0.0, 0.0));
 t.setRotation(tf::createQuaternionFromRPY(0.0, float(counter)/140.0, 0.0));
 br.sendTransform(tf::StampedTransform(t, time, "base_link", "rotating_frame"));

 counter++;
}
// %EndTag(frameCallback)%

// %Tag(processFeedback)%
void processFeedback( const visualization_msgs::InteractiveMarkerFeedbackConstPtr &feedback
)
{
 std::ostringstream s;
 s << "Feedback from marker '" << feedback->marker_name << "' "
           << " / control '" << feedback->control_name << "'";

 std::ostringstream mouse_point_ss;
 if( feedback->mouse_point_valid )
 {
           mouse_point_ss << " at " << feedback->mouse_point.x
           << ", " << feedback->mouse_point.y
           << ", " << feedback->mouse_point.z
```

```
            << " in frame " << feedback->header.frame_id;
  }

  switch ( feedback->event_type )
  {
            case visualization_msgs::InteractiveMarkerFeedback::BUTTON_CLICK:
            ROS_INFO_STREAM( s.str() << ": button click" << mouse_point_ss.str() << "." );
            break;

            case visualization_msgs::InteractiveMarkerFeedback::MENU_SELECT:
            ROS_INFO_STREAM( s.str() << ": menu item " << feedback->menu_entry_id << "
clicked" << mouse_point_ss.str() << "." );
            break;

            case visualization_msgs::InteractiveMarkerFeedback::POSE_UPDATE:
            ROS_INFO_STREAM( s.str() << ": pose changed"
            << "\nposition = "
            << feedback->pose.position.x
            << ", " << feedback->pose.position.y
            << ", " << feedback->pose.position.z
            << "\norientation = "
            << feedback->pose.orientation.w
            << ", " << feedback->pose.orientation.x
            << ", " << feedback->pose.orientation.y
            << ", " << feedback->pose.orientation.z
            << "\nframe: " << feedback->header.frame_id
            << " time: " << feedback->header.stamp.sec << "sec, "
            << feedback->header.stamp.nsec << " nsec" );
            break;

            case visualization_msgs::InteractiveMarkerFeedback::MOUSE_DOWN:
            ROS_INFO_STREAM( s.str() << ": mouse down" << mouse_point_ss.str() << "." );
            break;

            case visualization_msgs::InteractiveMarkerFeedback::MOUSE_UP:
            ROS_INFO_STREAM( s.str() << ": mouse up" << mouse_point_ss.str() << "." );
            break;
  }

  server->applyChanges();
}
// %EndTag(processFeedback)%

// %Tag(alignMarker)%
void alignMarker( const visualization_msgs::InteractiveMarkerFeedbackConstPtr &feedback )
{
  geometry_msgs::Pose pose = feedback->pose;

  pose.position.x = round(pose.position.x-0.5)+0.5;
```

```
  pose.position.y = round(pose.position.y-0.5)+0.5;

  ROS_INFO_STREAM( feedback->marker_name << ":"
                << " aligning position = "
                << feedback->pose.position.x
                << ", " << feedback->pose.position.y
                << ", " << feedback->pose.position.z
                << " to "
                << pose.position.x
                << ", " << pose.position.y
                << ", " << pose.position.z );

  server->setPose( feedback->marker_name, pose );
  server->applyChanges();
}
// %EndTag(alignMarker)%

double rand( double min, double max )
{
  double t = (double)rand() / (double)RAND_MAX;
  return min + t*(max-min);
}

void saveMarker( InteractiveMarker int_marker )
{
  server->insert(int_marker);
  server->setCallback(int_marker.name, &processFeedback);
}

////////////////////////////////////////////////////////////////////////////////

// %Tag(6DOF)%
void make6DofMarker( bool fixed, unsigned int interaction_mode, const tf::Vector3& position,
bool show_6dof )
{
  InteractiveMarker int_marker;
  int_marker.header.frame_id = "base_link";
  tf::pointTFToMsg(position, int_marker.pose.position);
  int_marker.scale = 1;

  int_marker.name = "simple_6dof";
  int_marker.description = "Simple 6-DOF Control";

  // insert a box
  makeBoxControl(int_marker);
  int_marker.controls[0].interaction_mode = interaction_mode;

  InteractiveMarkerControl control;
```

```
 if ( fixed )
 {
        int_marker.name += "_fixed";
        int_marker.description += "\n(fixed orientation)";
        control.orientation_mode = InteractiveMarkerControl::FIXED;
 }

 if (interaction_mode != visualization_msgs::InteractiveMarkerControl::NONE)
 {
        std::string mode_text;
        if( interaction_mode == visualization_msgs::InteractiveMarkerControl::MOVE_3D )
        mode_text = "MOVE_3D";
        if( interaction_mode == visualization_msgs::InteractiveMarkerControl::ROTATE_3D )
        mode_text = "ROTATE_3D";
        if( interaction_mode ==
visualization_msgs::InteractiveMarkerControl::MOVE_ROTATE_3D ) mode_text =
"MOVE_ROTATE_3D";
        int_marker.name += "_" + mode_text;
        int_marker.description = std::string("3D Control") + (show_6dof ? " + 6-DOF controls" :
"") + "\n" + mode_text;
 }

 if(show_6dof)
 {
        control.orientation.w = 1;
        control.orientation.x = 1;
        control.orientation.y = 0;
        control.orientation.z = 0;
        control.name = "rotate_x";
        control.interaction_mode = InteractiveMarkerControl::ROTATE_AXIS;
        int_marker.controls.push_back(control);
        control.name = "move_x";
        control.interaction_mode = InteractiveMarkerControl::MOVE_AXIS;
        int_marker.controls.push_back(control);

        control.orientation.w = 1;
        control.orientation.x = 0;
        control.orientation.y = 1;
        control.orientation.z = 0;
        control.name = "rotate_z";
        control.interaction_mode = InteractiveMarkerControl::ROTATE_AXIS;
        int_marker.controls.push_back(control);
        control.name = "move_z";
        control.interaction_mode = InteractiveMarkerControl::MOVE_AXIS;
        int_marker.controls.push_back(control);

        control.orientation.w = 1;
        control.orientation.x = 0;
        control.orientation.y = 0;
```

```
            control.orientation.z = 1;
            control.name = "rotate_y";
            control.interaction_mode = InteractiveMarkerControl::ROTATE_AXIS;
            int_marker.controls.push_back(control);
            control.name = "move_y";
            control.interaction_mode = InteractiveMarkerControl::MOVE_AXIS;
            int_marker.controls.push_back(control);
 }

 server->insert(int_marker);
 server->setCallback(int_marker.name, &processFeedback);
 if (interaction_mode != visualization_msgs::InteractiveMarkerControl::NONE)
            menu_handler.apply( *server, int_marker.name );
}
// %EndTag(6DOF)%

// %Tag(RandomDof)%
void makeRandomDofMarker( const tf::Vector3& position )
{
 InteractiveMarker int_marker;
 int_marker.header.frame_id = "base_link";
 tf::pointTFToMsg(position, int_marker.pose.position);
 int_marker.scale = 1;

 int_marker.name = "6dof_random_axes";
 int_marker.description = "6-DOF\n(Arbitrary Axes)";

 makeBoxControl(int_marker);

 InteractiveMarkerControl control;

 for ( int i=0; i<3; i++ )
 {
            control.orientation.w = rand(-1,1);
            control.orientation.x = rand(-1,1);
            control.orientation.y = rand(-1,1);
            control.orientation.z = rand(-1,1);
            control.interaction_mode = InteractiveMarkerControl::ROTATE_AXIS;
            int_marker.controls.push_back(control);
            control.interaction_mode = InteractiveMarkerControl::MOVE_AXIS;
            int_marker.controls.push_back(control);
 }

 server->insert(int_marker);
 server->setCallback(int_marker.name, &processFeedback);
}
// %EndTag(RandomDof)%
```

```
// %Tag(ViewFacing)%
void makeViewFacingMarker( const tf::Vector3& position )
{
 InteractiveMarker int_marker;
 int_marker.header.frame_id = "base_link";
 tf::pointTFToMsg(position, int_marker.pose.position);
 int_marker.scale = 1;

 int_marker.name = "view_facing";
 int_marker.description = "View Facing 6-DOF";

 InteractiveMarkerControl control;

 // make a control that rotates around the view axis
 control.orientation_mode = InteractiveMarkerControl::VIEW_FACING;
 control.interaction_mode = InteractiveMarkerControl::ROTATE_AXIS;
 control.orientation.w = 1;
 control.name = "rotate";

 int_marker.controls.push_back(control);

 // create a box in the center which should not be view facing,
 // but move in the camera plane.
 control.orientation_mode = InteractiveMarkerControl::VIEW_FACING;
 control.interaction_mode = InteractiveMarkerControl::MOVE_PLANE;
 control.independent_marker_orientation = true;
 control.name = "move";

 control.markers.push_back( makeBox(int_marker) );
 control.always_visible = true;

 int_marker.controls.push_back(control);

 server->insert(int_marker);
 server->setCallback(int_marker.name, &processFeedback);
}
// %EndTag(ViewFacing)%


// %Tag(Quadrocopter)%
void makeQuadrocopterMarker( const tf::Vector3& position )
{
 InteractiveMarker int_marker;
 int_marker.header.frame_id = "base_link";
 tf::pointTFToMsg(position, int_marker.pose.position);
 int_marker.scale = 1;

 int_marker.name = "quadrocopter";
 int_marker.description = "Quadrocopter";
```

```
  makeBoxControl(int_marker);

  InteractiveMarkerControl control;

  control.orientation.w = 1;
  control.orientation.x = 0;
  control.orientation.y = 1;
  control.orientation.z = 0;
  control.interaction_mode = InteractiveMarkerControl::MOVE_ROTATE;
  int_marker.controls.push_back(control);
  control.interaction_mode = InteractiveMarkerControl::MOVE_AXIS;
  int_marker.controls.push_back(control);

  server->insert(int_marker);
  server->setCallback(int_marker.name, &processFeedback);
}
// %EndTag(Quadrocopter)%

// %Tag(ChessPiece)%
void makeChessPieceMarker( const tf::Vector3& position )
{
  InteractiveMarker int_marker;
  int_marker.header.frame_id = "base_link";
  tf::pointTFToMsg(position, int_marker.pose.position);
  int_marker.scale = 1;

  int_marker.name = "chess_piece";
  int_marker.description = "Chess Piece\n(2D Move + Alignment)";

  InteractiveMarkerControl control;

  control.orientation.w = 1;
  control.orientation.x = 0;
  control.orientation.y = 1;
  control.orientation.z = 0;
  control.interaction_mode = InteractiveMarkerControl::MOVE_PLANE;
  int_marker.controls.push_back(control);

  // make a box which also moves in the plane
  control.markers.push_back( makeBox(int_marker) );
  control.always_visible = true;
  int_marker.controls.push_back(control);

  // we want to use our special callback function
  server->insert(int_marker);
  server->setCallback(int_marker.name, &processFeedback);

  // set different callback for POSE_UPDATE feedback
```

```
  server->setCallback(int_marker.name, &alignMarker,
visualization_msgs::InteractiveMarkerFeedback::POSE_UPDATE );
}
// %EndTag(ChessPiece)%

// %Tag(PanTilt)%
void makePanTiltMarker( const tf::Vector3& position )
{
  InteractiveMarker int_marker;
  int_marker.header.frame_id = "base_link";
  tf::pointTFToMsg(position, int_marker.pose.position);
  int_marker.scale = 1;

  int_marker.name = "pan_tilt";
  int_marker.description = "Pan / Tilt";

  makeBoxControl(int_marker);

  InteractiveMarkerControl control;

  control.orientation.w = 1;
  control.orientation.x = 0;
  control.orientation.y = 1;
  control.orientation.z = 0;
  control.interaction_mode = InteractiveMarkerControl::ROTATE_AXIS;
  control.orientation_mode = InteractiveMarkerControl::FIXED;
  int_marker.controls.push_back(control);

  control.orientation.w = 1;
  control.orientation.x = 0;
  control.orientation.y = 0;
  control.orientation.z = 1;
  control.interaction_mode = InteractiveMarkerControl::ROTATE_AXIS;
  control.orientation_mode = InteractiveMarkerControl::INHERIT;
  int_marker.controls.push_back(control);

  server->insert(int_marker);
  server->setCallback(int_marker.name, &processFeedback);
}
// %EndTag(PanTilt)%

// %Tag(Menu)%
void makeMenuMarker( const tf::Vector3& position )
{
  InteractiveMarker int_marker;
  int_marker.header.frame_id = "base_link";
  tf::pointTFToMsg(position, int_marker.pose.position);
  int_marker.scale = 1;
```

```
  int_marker.name = "context_menu";
  int_marker.description = "Context Menu\n(Right Click)";

  InteractiveMarkerControl control;

  control.interaction_mode = InteractiveMarkerControl::MENU;
  control.name = "menu_only_control";

  Marker marker = makeBox( int_marker );
  control.markers.push_back( marker );
  control.always_visible = true;
  int_marker.controls.push_back(control);

  server->insert(int_marker);
  server->setCallback(int_marker.name, &processFeedback);
  menu_handler.apply( *server, int_marker.name );
}
// %EndTag(Menu)%

// %Tag(Button)%
void makeButtonMarker( const tf::Vector3& position )
{
  InteractiveMarker int_marker;
  int_marker.header.frame_id = "base_link";
  tf::pointTFToMsg(position, int_marker.pose.position);
  int_marker.scale = 1;

  int_marker.name = "button";
  int_marker.description = "Button\n(Left Click)";

  InteractiveMarkerControl control;

  control.interaction_mode = InteractiveMarkerControl::BUTTON;
  control.name = "button_control";

  Marker marker = makeBox( int_marker );
  control.markers.push_back( marker );
  control.always_visible = true;
  int_marker.controls.push_back(control);

  server->insert(int_marker);
  server->setCallback(int_marker.name, &processFeedback);
}
// %EndTag(Button)%

// %Tag(Moving)%
void makeMovingMarker( const tf::Vector3& position )
{
  InteractiveMarker int_marker;
```

```
  int_marker.header.frame_id = "moving_frame";
  tf::pointTFToMsg(position, int_marker.pose.position);
  int_marker.scale = 1;

  int_marker.name = "moving";
  int_marker.description = "Marker Attached to a\nMoving Frame";

  InteractiveMarkerControl control;

  control.orientation.w = 1;
  control.orientation.x = 1;
  control.orientation.y = 0;
  control.orientation.z = 0;
  control.interaction_mode = InteractiveMarkerControl::ROTATE_AXIS;
  int_marker.controls.push_back(control);

  control.interaction_mode = InteractiveMarkerControl::MOVE_PLANE;
  control.always_visible = true;
  control.markers.push_back( makeBox(int_marker) );
  int_marker.controls.push_back(control);

  server->insert(int_marker);
  server->setCallback(int_marker.name, &processFeedback);
}
// %EndTag(Moving)%

// %Tag(main)%
int main(int argc, char** argv)
{
  ros::init(argc, argv, "basic_controls");
  ros::NodeHandle n;

  // create a timer to update the published transforms
  ros::Timer frame_timer = n.createTimer(ros::Duration(0.01), frameCallback);

  server.reset( new interactive_markers::InteractiveMarkerServer("basic_controls","",false) );

  ros::Duration(0.1).sleep();

  menu_handler.insert( "First Entry", &processFeedback );
  menu_handler.insert( "Second Entry", &processFeedback );
  interactive_markers::MenuHandler::EntryHandle sub_menu_handle =
menu_handler.insert("Submenu" );
  menu_handler.insert( sub_menu_handle, "First Entry", &processFeedback );
  menu_handler.insert( sub_menu_handle, "Second Entry", &processFeedback );

  tf::Vector3 position;
  position = tf::Vector3(-3, 3, 0);
  make6DofMarker( false, visualization_msgs::InteractiveMarkerControl::NONE, position, true );
```

```
  position = tf::Vector3( 0, 3, 0);
  make6DofMarker( true, visualization_msgs::InteractiveMarkerControl::NONE, position, true );
  position = tf::Vector3( 3, 3, 0);
  makeRandomDofMarker( position );
  position = tf::Vector3(-3, 0, 0);
  make6DofMarker( false, visualization_msgs::InteractiveMarkerControl::ROTATE_3D, position,
false );
  position = tf::Vector3( 0, 0, 0);
  make6DofMarker( false, visualization_msgs::InteractiveMarkerControl::MOVE_ROTATE_3D,
position, true );
  position = tf::Vector3( 3, 0, 0);
  make6DofMarker( false, visualization_msgs::InteractiveMarkerControl::MOVE_3D, position,
false );
  position = tf::Vector3(-3,-3, 0);
  makeViewFacingMarker( position );
  position = tf::Vector3( 0,-3, 0);
  makeQuadrocopterMarker( position );
  position = tf::Vector3( 3,-3, 0);
  makeChessPieceMarker( position );
  position = tf::Vector3(-3,-6, 0);
  makePanTiltMarker( position );
  position = tf::Vector3( 0,-6, 0);
  makeMovingMarker( position );
  position = tf::Vector3( 3,-6, 0);
  makeMenuMarker( position );
  position = tf::Vector3( 0,-9, 0);
  makeButtonMarker( position );

  server->applyChanges();

  ros::spin();

  server.reset();
}
```

Replace this in CMakeLists.txt

```
add_executable(basic_controls src/basic_controls.cpp)
target_link_libraries(basic_controls ${catkin_LIBRARIES})
```

Replace find package function with

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  visualization_msgs
  interactive_markers
  tf
```

)

cd ~/ name of your workspace

catkin_make

source devel/setup.bash

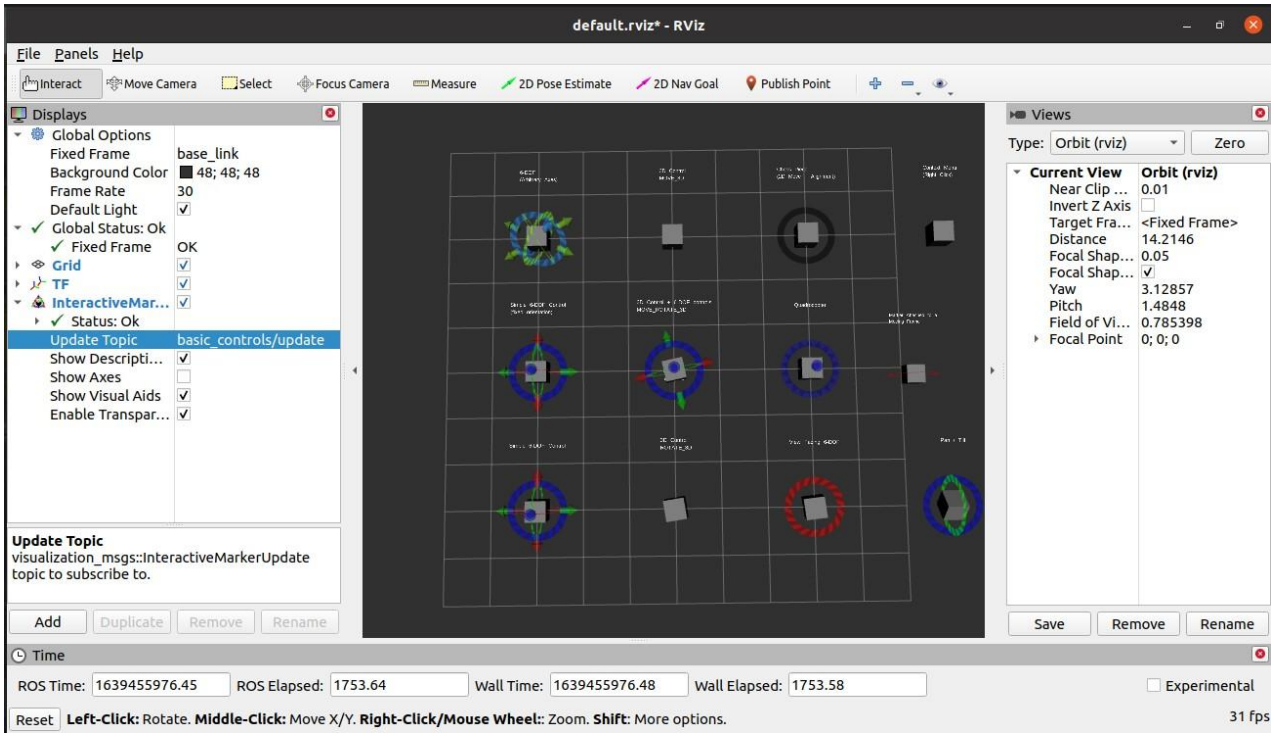
rosrun using_markers basic_controls

rosrun rviz rviz(new terminal)

In the rviz , set the Fixed Frame field to base_link

Add Interactive markers to display

In interactive markers, set update Topic to basic_controls/update

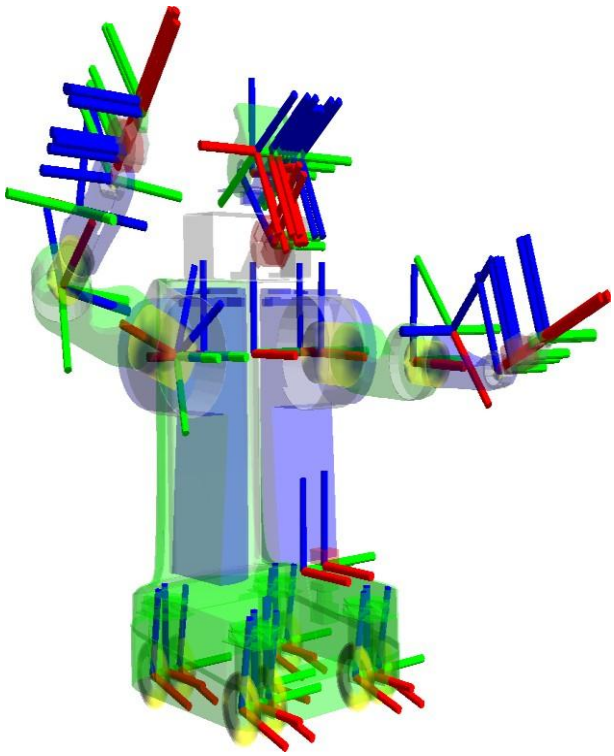


EXPERIMENT:4

AIM: Introduction to tf -- broadcast the state of a robot to tf, get access to frame transformations, adding a frame, wait For Transform function, setting up your robot using tf, publish the state of your robot to tf, using the robot state publisher.

Introduction to tf

- tf is a package that lets the user **keep track of multiple frames** over time.
- ft. maintains the **relationship between frames** in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

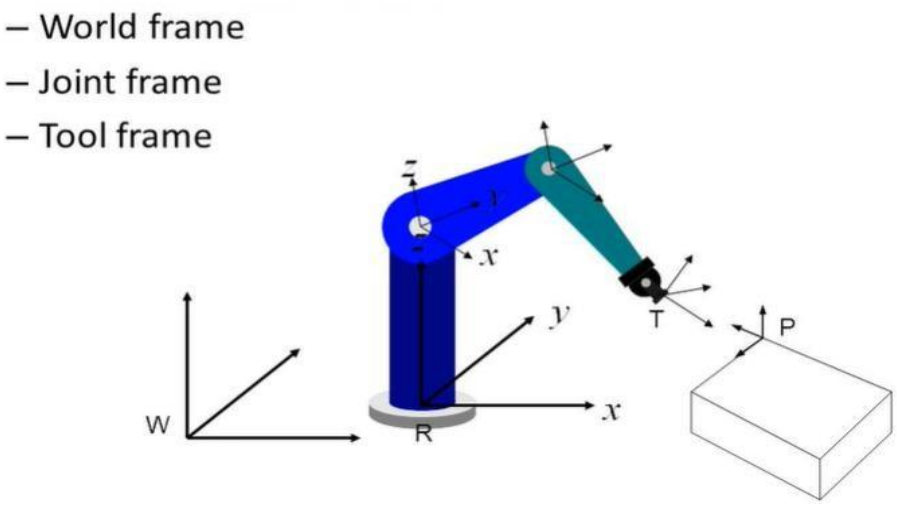


Frame: Frames in a robot **define a coordinate system** that the robot uses to know where it is and where to go.

A frame is comprised of six main components: an X, Y, & Z axis and a rotation about each of these axes.

World frame is the ultimate reference frame. Directly or indirectly, **all other frames are defined with respect to the World frame**. It is a **fixed frame**.

Pose: The combination **of position and orientation** is referred to as the pose of an object



What does tf do? Why should use tf?

A robotic system typically has many 3D coordinate frames that change over **time**, such as a world frame, base frame, gripper frame, head frame, etc. tf keeps track of all these frames over time, and allows you to ask questions like:

- Where was the head frame relative to the world frame, 5 seconds ago?
- What is the pose of the object in my gripper relative to my base?
- What is the current pose of the base frame in the map frame?

These informations we got from TF

tf tools

1.Using rqt_tf_tree

rqt_tf_tree is a runtime tool for visualizing the tree of frames being broadcast over ROS

rosrun rqt_tf_tree rqt_tf_tree

2. Using tf_echo

tf_echo reports the transform between any two frames broadcast over ROS.

rosrun tf tf_echo [reference_frame] [target_frame]

3. rviz and tf

rviz is a visualization tool that is useful for examining tf frames.

rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz

STEPS TO BE FOLLWED

1.Create tf package

1. Package must be created inside **src** folder of your workspace

cd name of your workspace

cd src

catkin_create_pkg learning_tf tf roscpp rospy turtlesim

create a package called learning_tf that depends on tf, roscpp, rospy and turtlesim

2. Build your package

(Go to your workspace and run catkin_make)

cd name of your workspace

catkin_make

source devel/setup.bash

2.How to broadcast transforms

This will give an idea about how to broadcast coordinate frames to tf. In this case, we want to broadcast the changing coordinate frames of the turtles, as they move around.

1. Go to the package created

roscd learning_tf

2. Make a new directory called **nodes** in learning_tf package.

mkdir nodes
cd nodes

Fire up your editor and paste the following code into a new file called turtle_tf_broadcaster.py in the directory called nodes.

gedit turtle_tf_broadcaster.py (open a text editor named turtle_tf_broadcaster.py)

```
#!/usr/bin/env python3
import roslib//command lines for message generation
roslib.load_manifest('learning_tf') //reads the package manifest and sets up the python library path based on the package dependencies.
import rospy
import tf
```

```
import turtlesim.msg

def handle_turtle_pose(msg, turtlename):
    br = tf.TransformBroadcaster()
    br.sendTransform((msg.x, msg.y, 0),
                     tf.transformations.quaternion_from_euler(0, 0, msg.theta),
                     rospy.Time.now(),
                     turtlename,
                     "world")

if __name__ == '_main_':
    rospy.init_node('turtle_tf_broadcaster')
    turtlename = rospy.get_param('~turtle')
    rospy.Subscriber('/%s/pose' % turtlename,
                     turtlesim.msg.Pose,
                     handle_turtle_pose,
                     turtlename)
    rospy.spin()
```

3. To make the node executable:

chmod +x turtle_tf_broadcaster.py

4. Running the broadcaster

Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters.

Note: roslaunch will also start roscore if no master has been set. Pushing Ctrl-C in a terminal with a launch file running will close all nodes that were started with that launch files.

1. Create a directory launch in the learning_tf Package.

roscd learning_tf

mkdir launch
cd launch

2. Create a file **start_demo.launch** in the launch directory using gedit text editor and copy the given code

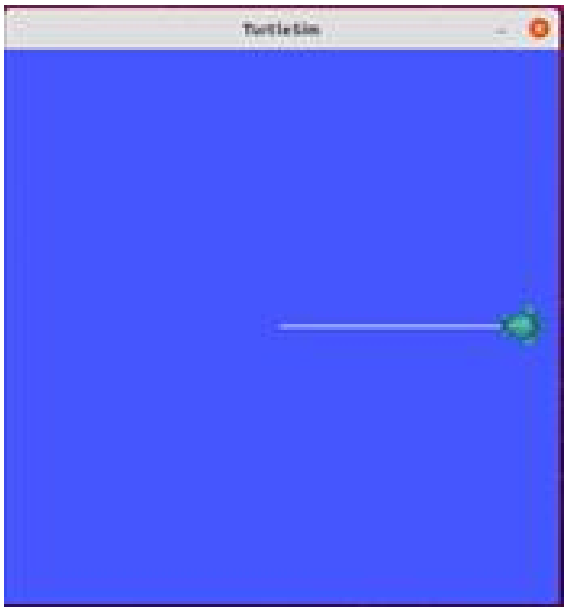
gedit start_demo.launch (open a text editor named start_demo.launch)

```
<launch>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <node name="turtle1_tf_broadcaster" pkg="learning_tf" type="turtle_tf_broadcaster.py" respawn="false" output="screen" >
    <param name="turtle" type="string" value="turtle1" />
  </node>
  <node name="turtle2_tf_broadcaster" pkg="learning_tf" type="turtle_tf_broadcaster.py" respawn="false" output="screen" >
    <param name="turtle" type="string" value="turtle2" />
  </node>

</launch>
```

3. To run the turtle broadcaster demo

source devel/setup.bash (must be in the workspace)

roslaunch learning_tf start_demo.launch



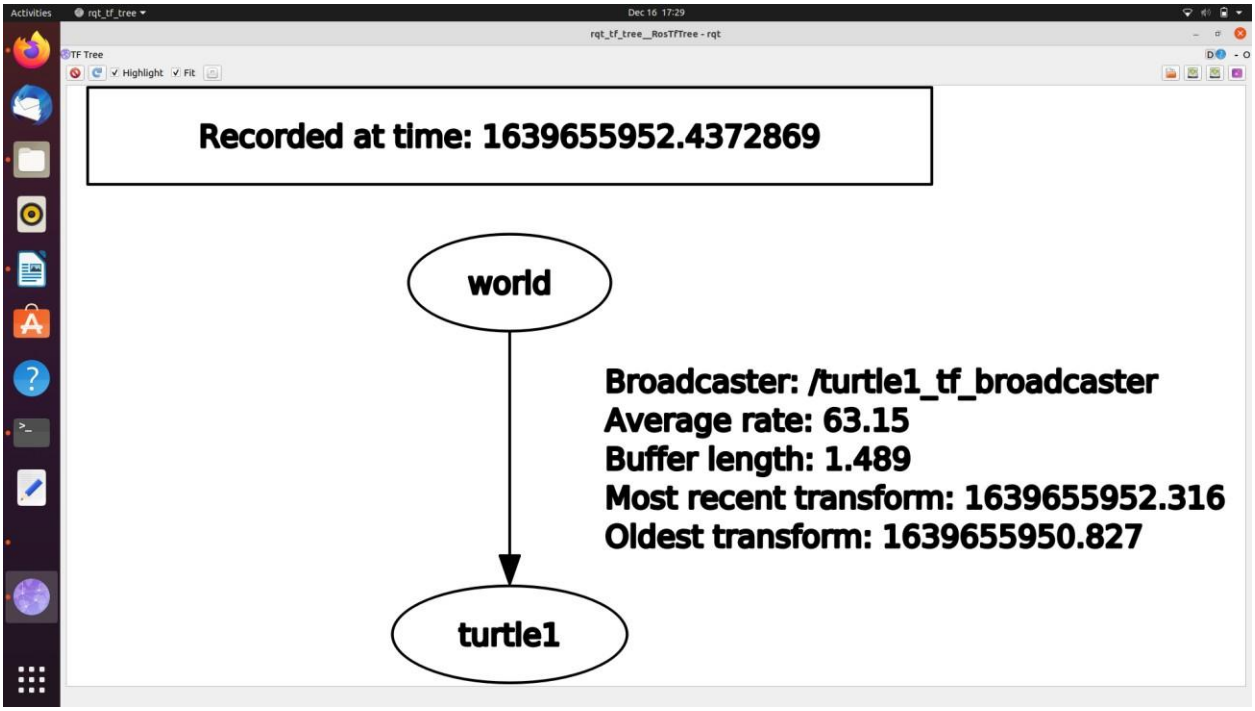
4. Checking the results

tf Tools

1. Using rqt_tf_tree

rqt_tf_tree is a runtime tool for visualizing the tree of frames being broadcast over ROS.

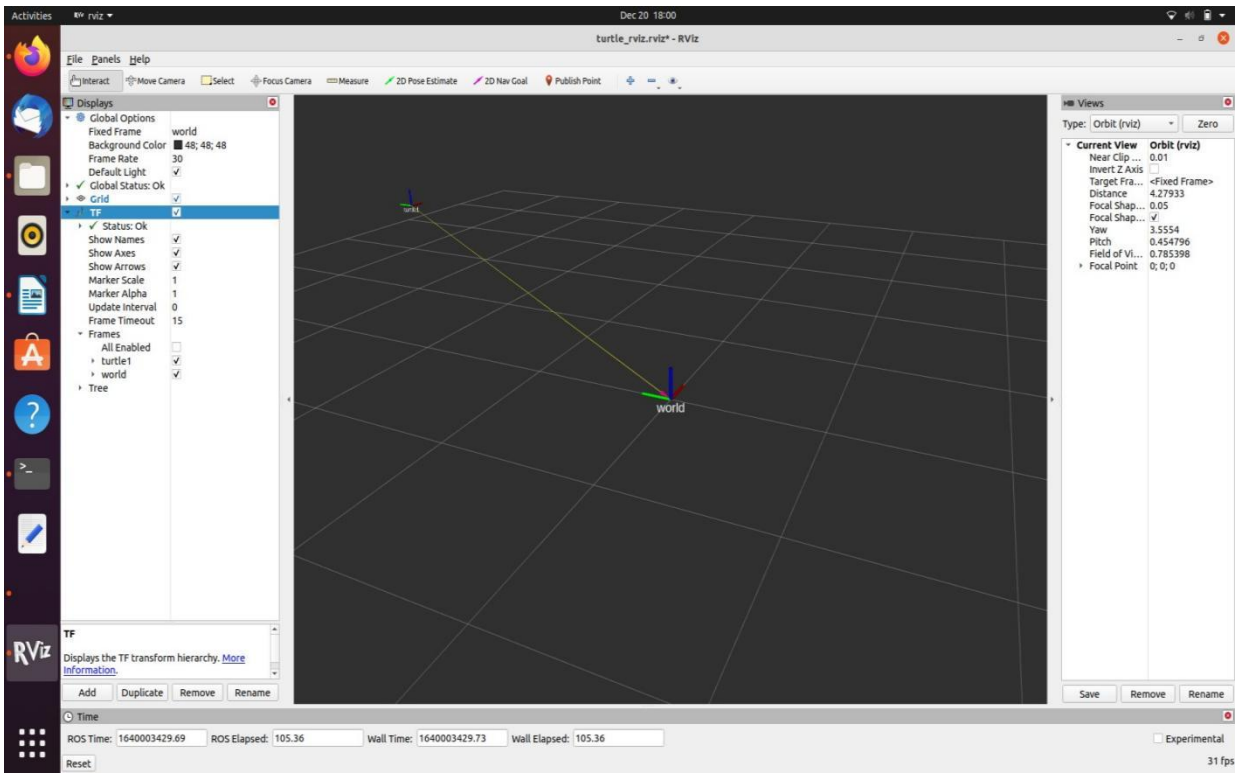
rosrun rqt_tf_tree rqt_tf_tree (new terminal)



2. rviz and tf

rviz is a visualization tool that is useful for examining tf frames.

rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz (new terminal)

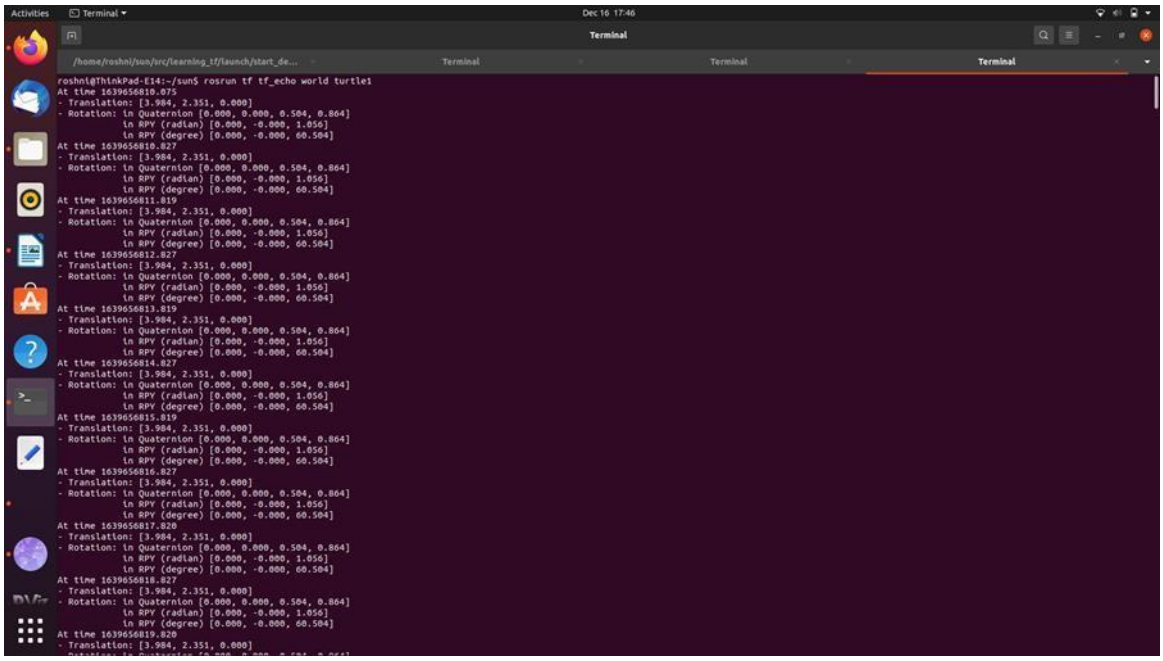


3. Using tf_echo

tf_echo reports the transform between any two frames broadcast over ROS

Print information about a particular transformation between a source_frame and a target_frame

rosrun tf tf_echo world turtle1 (new terminal)



RPY -Any three-dimensional rotation can be described as a sequence of yaw, pitch, and roll rotations

A *yaw* is a counterclockwise rotation of about the -axis

A *pitch* is a counterclockwise rotation of β about the y-axis.
A *roll* is a counterclockwise rotation of about the -axis.

3.Writing a tf listener

This section will give an idea about how to use tf to get access to frame transformations

1.Create a tf listener

first create the source files. Go to the tf package

roscd learning_tf

cd nodes

gedit turtle_tf_listener.py

Paste the given code to turtle_tf_listener.py text editor and save it

turtle_tf_listener.py

```
#!/usr/bin/env python3
import roslib
roslib.load_manifest('learning_tf')
import rospy
import math
import tf
import geometry_msgs.msg
import turtlesim.srv

if__name__== '_main_':
    rospy.init_node('turtle_tf_listener')

    listener = tf.TransformListener()

    rospy.wait_for_service('spawn')
    spawner = rospy.ServiceProxy('spawn', turtlesim.srv.Spawn)
    spawner(4, 2, 0, 'turtle2')

    turtle_vel = rospy.Publisher('turtle2/cmd_vel', geometry_msgs.msg.Twist,queue_size=1)

    rate = rospy.Rate(10.0)
```

```
while not rospy.is_shutdown():
    try:
        (trans,rot) = listener.lookupTransform('/turtle2', '/turtle1', rospy.Time(0))
    except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
        continue

    angular = 4 * math.atan2(trans[1], trans[0])
    linear = 0.5 * math.sqrt(trans[0] ** 2 + trans[1] ** 2)
    cmd = geometry_msgs.msg.Twist()
    cmd.linear.x = linear
    cmd.angular.z = angular
    turtle_vel.publish(cmd)

    rate.sleep()
```

2. To make the node executable:

chmod +x turtle_tf_listener.py

3. Run the listener

a. Open the launch file called **start_demo.launch** and add the following lines:

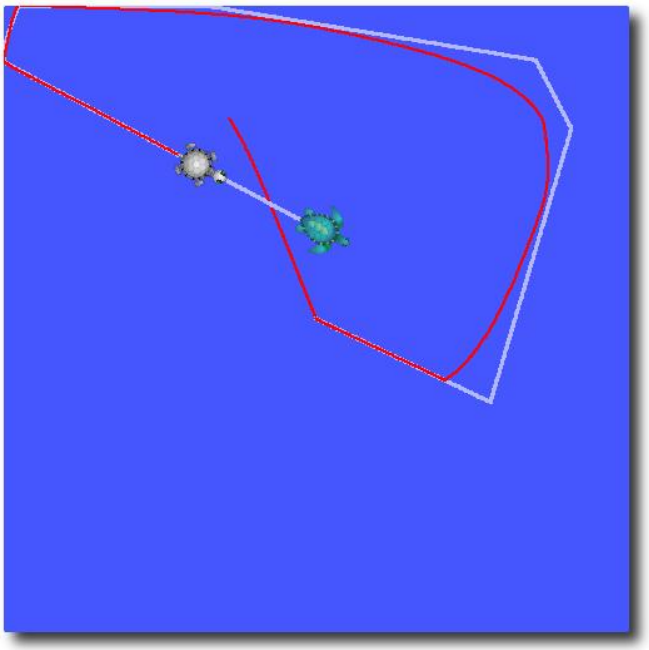
(Path:-Home folder-your work space-SRC- learning_tf-launch- start_demo.launch)
First, make sure you stopped the launch file from the previous program (use Ctrl-c).

```
<launch>
  ...
  <node pkg="learning_tf" type="turtle_tf_listener.py"
        name="listener" />
</launch>
```

b. To run the node

source devel/setup.bash (must be in the workspace)

roslaunch learning_tf start_demo.launch

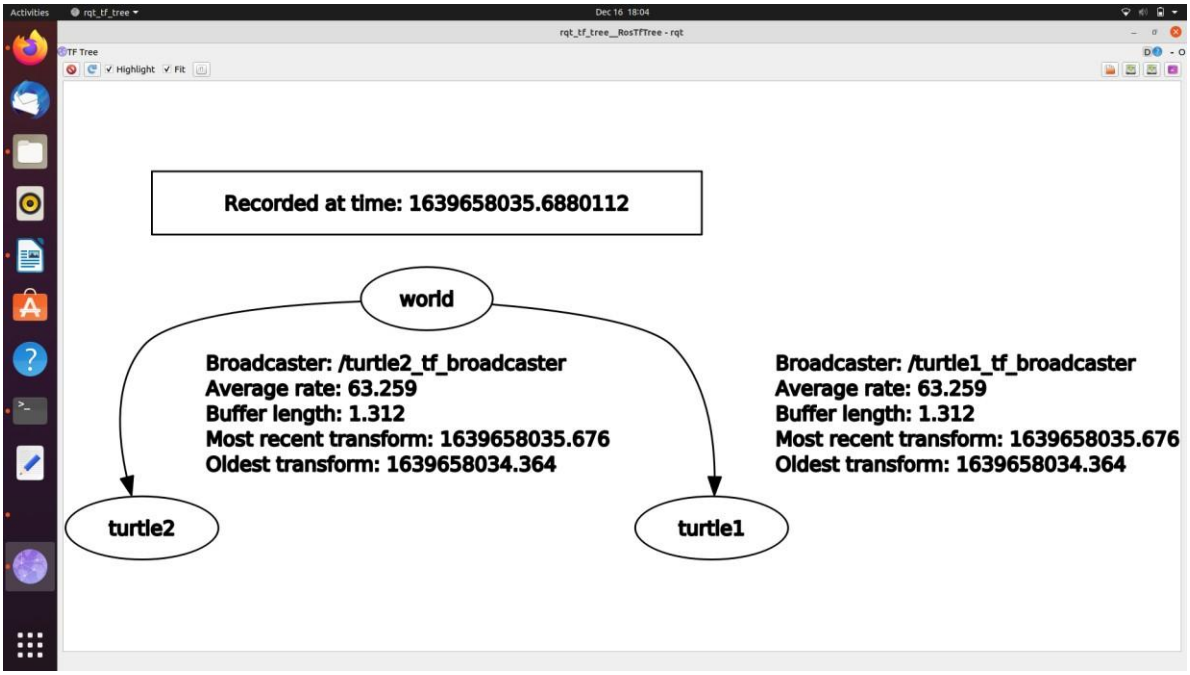


4. Checking the results

tf Tools

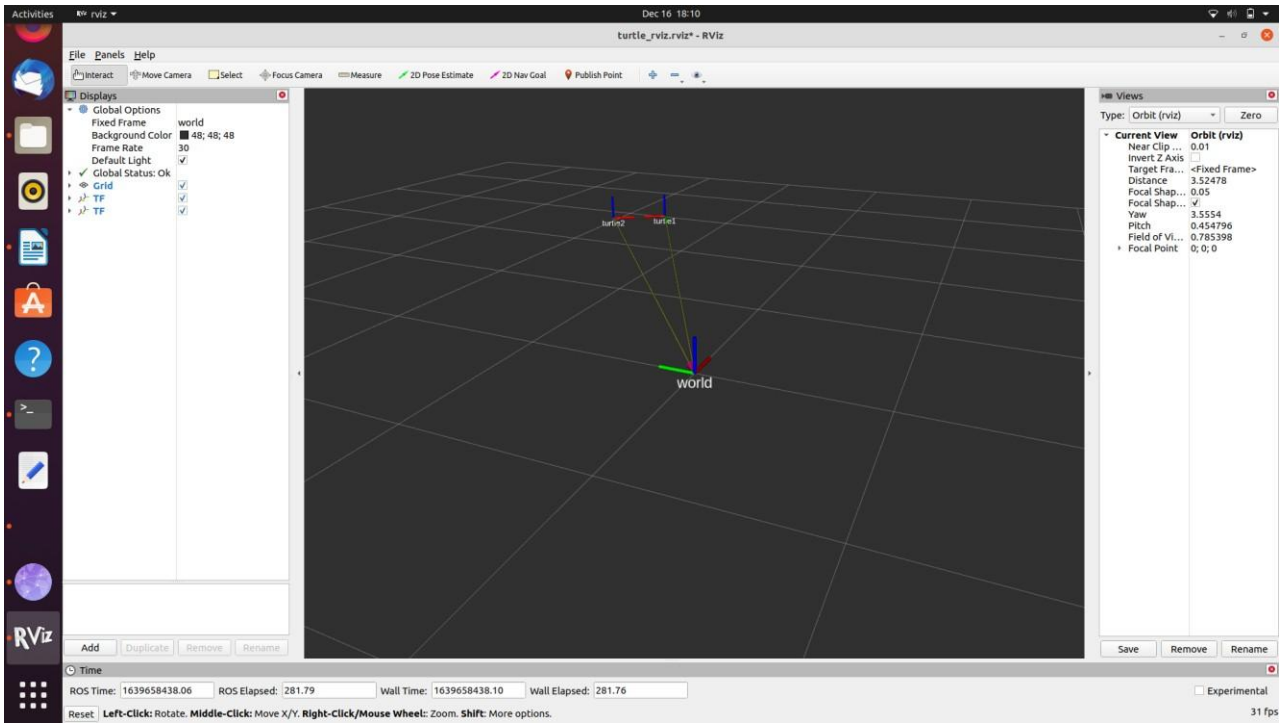
1. Using rqt_tf_tree

rosrun rqt_tf_tree rqt_tf_tree (new terminal)



2.rviz and tf

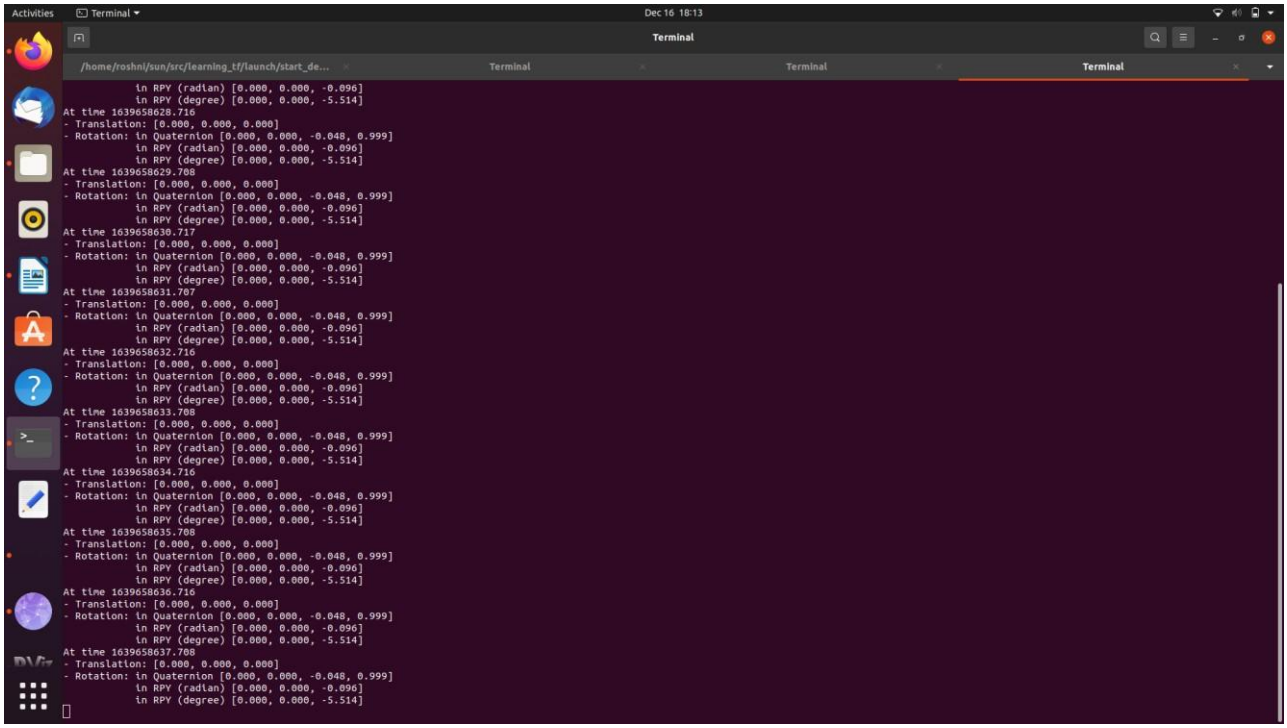
rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz (new terminal)



2.Using tf_echo

rosrun tf tf_echo turtle1 turtle2 (new terminal)

Let's look at the transform of the turtle2 frame with respect to turtle1 frame which is equivalent to the product of the transform from turtle1 to the world multiplied by the transform from the world to turtle2 frame.



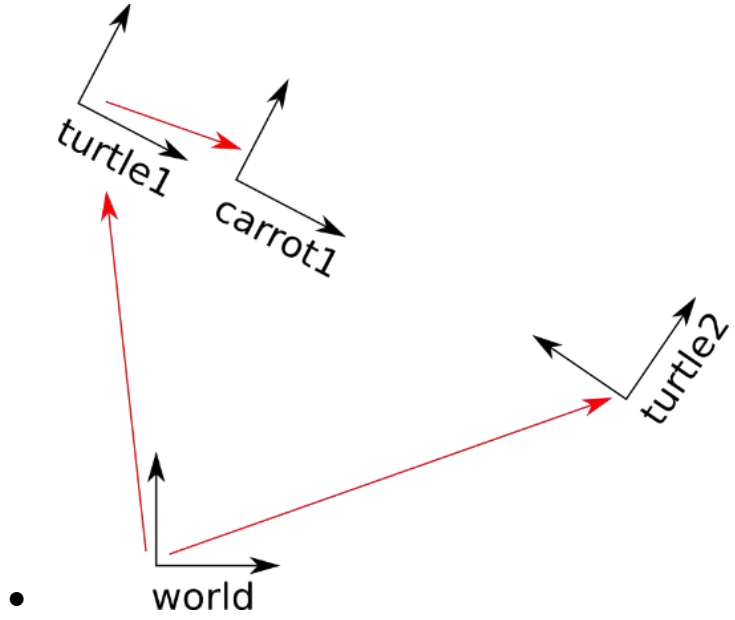
4. Adding a frame

Why adding frames

For many tasks it is easier to think inside a local frame, e.g. it is easier to reason about a laser scan in a frame at the center of the laser scanner. tf allows you to define a local frame for each sensor, link, etc in your system. And tf will take care of all the extra frame transforms that are introduced.

Where to add frames

tf builds up a **tree structure** of frames; it does not allow a closed loop in the frame structure. This means that a frame only has one single parent, but it can have multiple children. Currently our tf tree contains three frames: world, turtle1 and turtle2. The two turtles are children of world. If we want to add a new frame to tf, one of the three existing frames needs to be the parent frame, and the new frame will become a child frame.



4a. How to add a fixed frame

In our turtle example, we'll add a new fixed frame to the first turtle. This frame will be the "carrot" for the second turtle.

1. Create the source files. Go to the **tf package**

roscd learning_tf

2. Fire up gedit text editor and paste the following code into a file called fixed_tf_broadcaster.py

cd nodes

gedit fixed_tf_broadcaster.py

fixed_tf_broadcaster.py

```
#!/usr/bin/env python3
import roslib
roslib.load_manifest('learning_tf')

import rospy
import tf

if__name__== '_main_':
   rospy.init_node('fixed_tf_broadcaster')
```

```
br = tf.TransformBroadcaster()
rate = rospy.Rate(10.0)
while not rospy.is_shutdown():
    br.sendTransform((0.0, 2.0, 0.0),
                     (0.0, 0.0, 0.0, 1.0),
                     rospy.Time.now(),
                     "carrot1",
                     "turtle1")
    rate.sleep()
```

3. To make the node executable:

chmod +x fixed_tf_broadcaster.py

4.Running the frame broadcaster

Edit the **start_demo.launch** launch file. Simply add the following line:

First, make sure you stopped the launch file from the previous program (use Ctrl-c).

```
<launch>
  ...
  <node pkg="learning_tf" type="fixed_tf_broadcaster.py"
        name="broadcaster_fixed" />
</launch>
```

source devel/setup.bash (must be in the workspace)

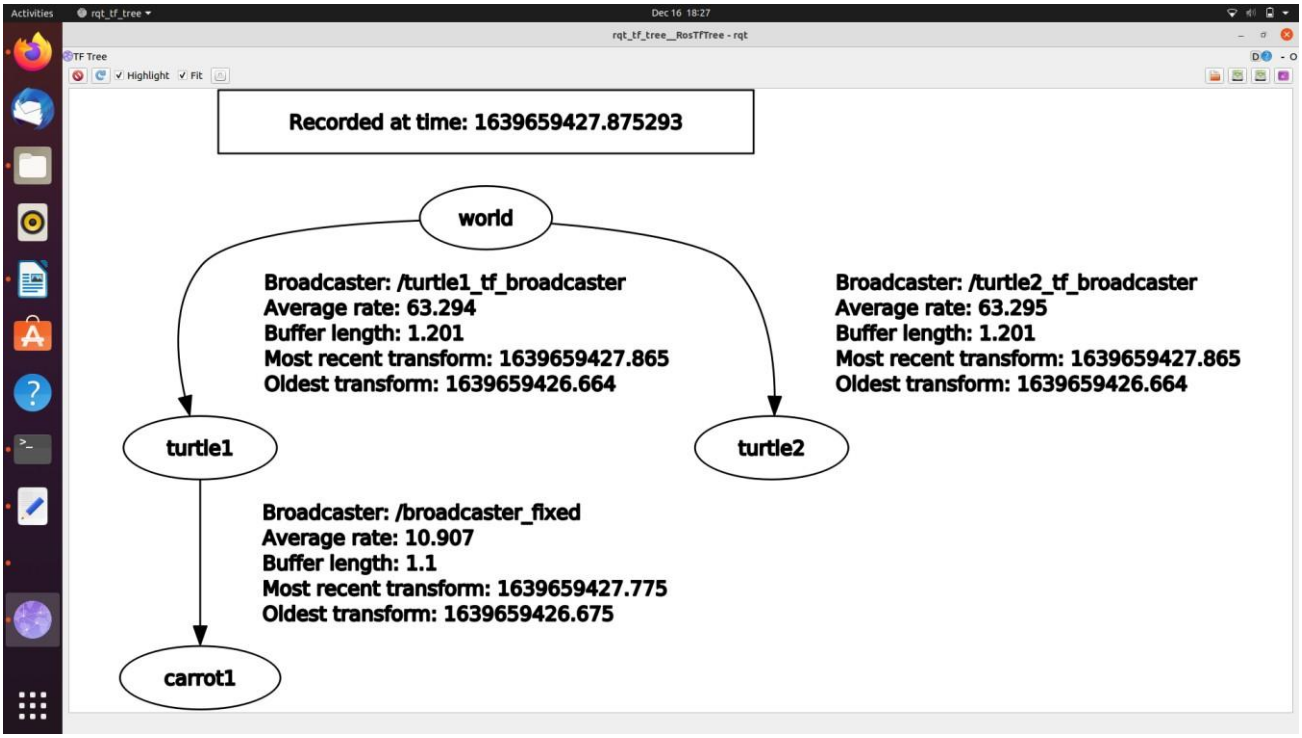
roslaunch learning_tf start_demo.launch

4. Checking the results

tf Tools

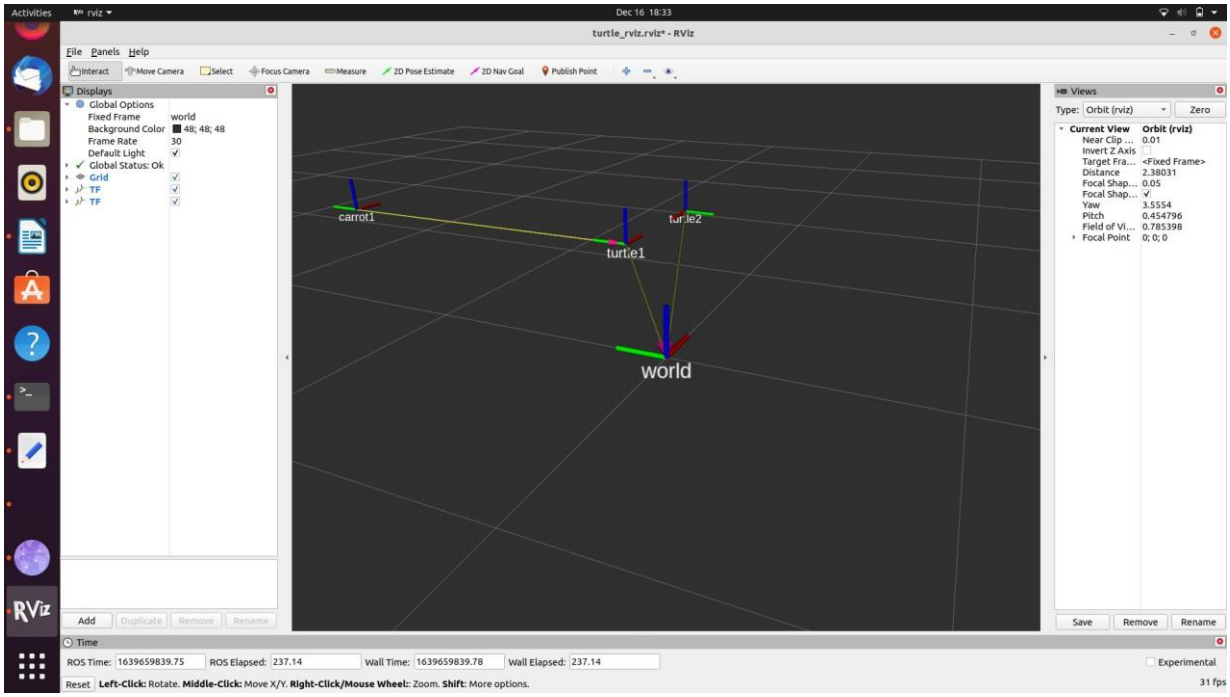
1. Using rqt_tf_tree

rosrun rqt_tf_tree rqt_tf_tree (new terminal)



2.rviz and tf

rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz (new terminal)



from RVIZ we can easily understand that a new fixed frame is added to the first turtle

4b. Change the behavior of the listener.

In listener node we set turtle2 follow turtle 1. Here we can change the behavior of the listener.Here we will set turtle 2 follow carrot1 frame

1.Open the **nodes/turtle_tf_listener.py** file, and simple replace "/turtle1" with "/carrot1": like in the line and save it

turtle_tf_listener.py

```
#!/usr/bin/env python3
import roslib
roslib.load_manifest('learning_tf')
import rospy
import math
import tf
import geometry_msgs.msg
import turtlesim.srv

if __name__ == '__main__':
    rospy.init_node('turtle_tf_listener')
```

```
listener = tf.TransformListener()

rospy.wait_for_service('spawn')
spawner = rospy.ServiceProxy('spawn', turtlesim.srv.Spawn)
spawner(4, 2, 0, 'turtle2')

turtle_vel = rospy.Publisher('turtle2/cmd_vel', geometry_msgs.msg.Twist,queue_size=1)

rate = rospy.Rate(10.0)
while not rospy.is_shutdown():
    try:
        (trans,rot) = listener.lookupTransform('/turtle2', '/turtle1', rospy.Time(0))
    except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
        continue

    angular = 4 * math.atan2(trans[1], trans[0])
    linear = 0.5 * math.sqrt(trans[0] ** 2 + trans[1] ** 2)
    cmd = geometry_msgs.msg.Twist()
    cmd.linear.x = linear
    cmd.angular.z = angular
    turtle_vel.publish(cmd)

    rate.sleep()
```

2. Run the node

source devel/setup.bash (must be in the workspace)

roslaunch learning_tf start_demo.launch

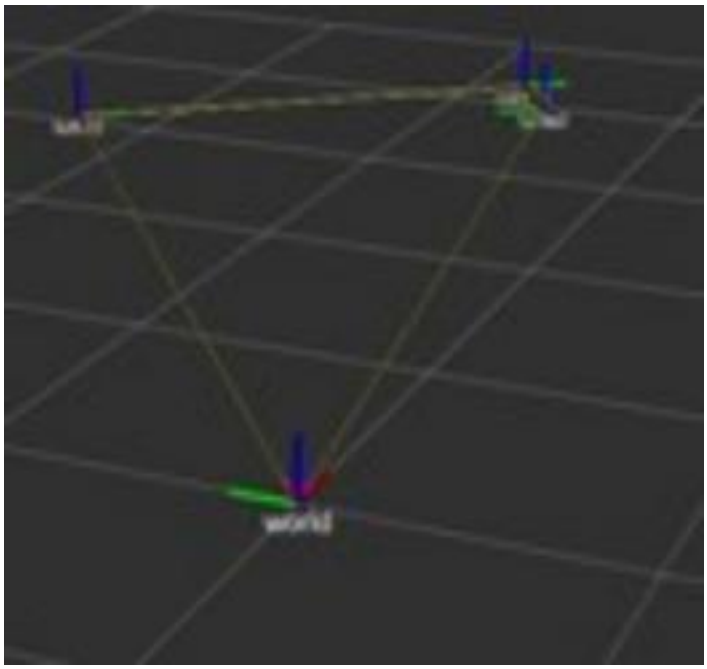
And now the good part: just restart the turtle demo, and you'll see the second turtle following the carrot instead of the first turtle! Remember that the carrot is 2 meters to the left of turtle1. There is no visual representation for the carrot, but you should see the second turtle moving to that point.

tf Tools

1.rviz and tf

rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz (new terminal)

By using RVIZ visualization tool you can easily understood that our turtle2 follow carrot1 frame.



4c. Broadcasting a moving frame

The extra frame we published in the above program is a fixed frame that doesn't change over time in relation to the parent frame. However, if you want to publish a moving frame you can code the broadcaster to change the frame over time. Let's change our carrot1 frame to change relative to turtle1 over time.

1.Create the file dynamic_tf_broadcaster.py in directory nodes

roscd learning_tf

cd nodes

gedit dynamic_tf_broadcaster.py

dynamic_tf_broadcaster.py

```
#!/usr/bin/env python3
import roslib
roslib.load_manifest('learning_tf')

import rospy
import tf
import math

if__name__== '_main_':
   rospy.init_node('dynamic_tf_broadcaster')
   br = tf.TransformBroadcaster()
   rate = rospy.Rate(10.0)
   while not rospy.is_shutdown():
      t = rospy.Time.now().to_sec() * math.pi
      br.sendTransform((2.0 * math.sin(t), 2.0 * math.cos(t), 0.0),
                       (0.0, 0.0, 0.0, 1.0),
                       rospy.Time.now(),
                       "carrot1",
                       "turtle1")
      rate.sleep()
```

Note that instead of defining a fixed offset from turtle1, we are using a sin and cos function based on the current time to cause the definition of the frame's offset.

2. To make the node executable:

chmod +x dynamic_tf_broadcaster.py

3. Open the nodes/turtle_tf_listener.py file, and simple replace "/carrot1" with "/turtle1": like in the line and save it

```
(trans,rot) = listener.lookupTransform("/turtle2", "/turtle1", rospy.Time(0))
```

4. To test this code, remember to change the launch file to point to our new, dynamic broadcaster for the definition of carrot1 instead of the fixed broadcaster above:

First, make sure you stopped the launch file from the previous program (use Ctrl-c).

```
<launch>
  ...
  <node pkg="learning_tf" type="dynamic_tf_broadcaster.py"
        name="broadcaster_dynamic" />
</launch>
```

source devel/setup.bash (must be in the workspace)

roslaunch learning_tf start_demo.launch

4. Checking the results

tf Tools

1.rviz and tf

rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz (new terminal)

By using RVIZ visualization tool you can easily understood that our carrot1 frame change relative to turtle1 over time.

5.Wait For Transform function

tf provides a nice tool that will wait until a transform becomes available.

Instead of making the second turtle go to where the first turtle is **now**, make the second turtle go to where the first turtle was **5 seconds ago**.

1 Edit **turtle_tf_listener.py** program.with the given code

```
try:
    now = rospy.Time.now() - rospy.Duration(5.0)
    listener.waitForTransform("/turtle2", "/turtle1", now, rospy.Duration(1.0))
    (trans, rot) = listener.lookupTransform("/turtle2", "/turtle1", now)
except (tf.Exception, tf.LookupException, tf.ConnectivityException):
```

turtle_tf_listener.py

```
#!/usr/bin/env python3
import roslib
roslib.load_manifest('learning_tf')
import rospy
import math
import tf
import geometry_msgs.msg
import turtlesim.srv

if__name__== '_main_':
    rospy.init_node('turtle_tf_listener')

    listener = tf.TransformListener()

    rospy.wait_for_service('spawn')
    spawner = rospy.ServiceProxy('spawn', turtlesim.srv.Spawn)
    spawner(4, 2, 0, 'turtle2')
```

```
turtle_vel = rospy.Publisher('turtle2/cmd_vel', geometry_msgs.msg.Twist,queue_size=1)

rate = rospy.Rate(10.0)
while not rospy.is_shutdown():
    try:
        (trans,rot) = listener.lookupTransform('/turtle2', '/turtle1', rospy.Time(0))
    except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
        continue

    angular = 4 * math.atan2(trans[1], trans[0])
    linear = 0.5 * math.sqrt(trans[0] ** 2 + trans[1] ** 2)
    cmd = geometry_msgs.msg.Twist()
    cmd.linear.x = linear
    cmd.angular.z = angular
    turtle_vel.publish(cmd)

    rate.sleep()
```

waitForTransform() takes four arguments:

a Wait for the transform from this frame...
b ... to this frame,
c at this time, and
d timeout: don't wait for longer than this maximum duration

So waitForTransform() will actually block until the transform between the two turtles becomes available (this will usually take a few milli-seconds), OR --if the transform does not become available-- until the timeout has been reached.

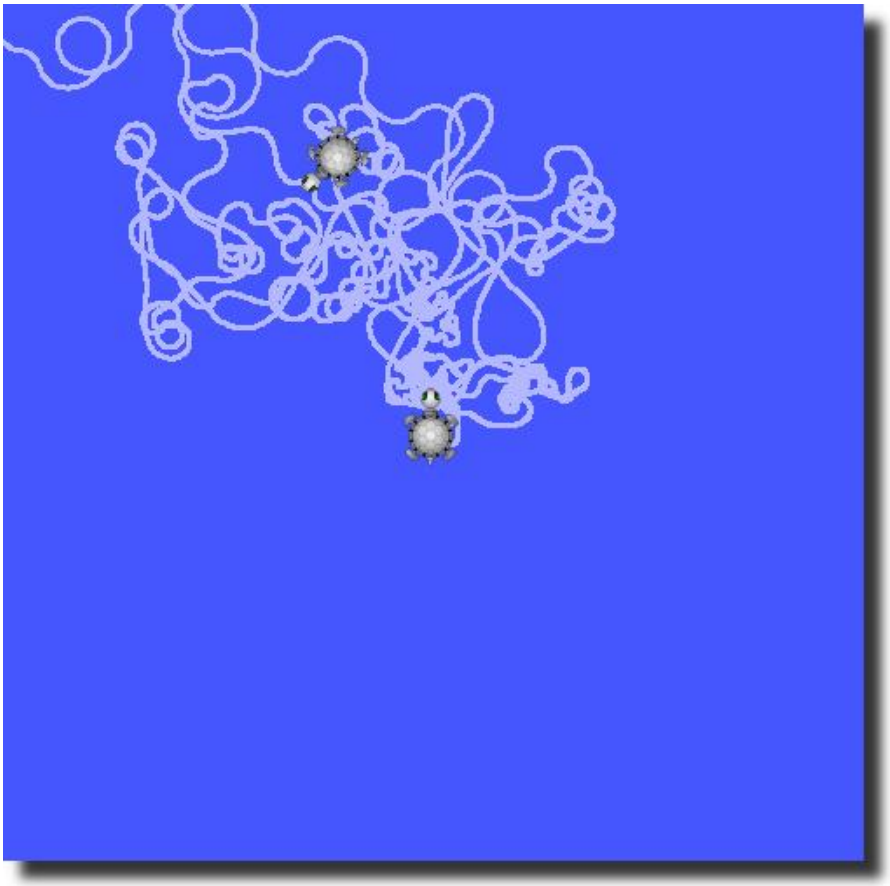
if you would run this, what would you expect to see? Definitely during the first 5 seconds the second turtle would not know where to go, because we do not yet have a 5 second history of the first turtle

2 Run the node

catkin_make(must be in the workspace)

source devel/setup.bash

roslaunch learning_tf start_demo.launch



Is your turtle driving around uncontrollably like in this screenshot? So what is happening?

- We asked tf, *"What was the pose of /turtle1 5 seconds ago, relative to /turtle2 5 seconds ago?"*. This means we are controlling the second turtle based on where it was 5 seconds ago as well as where the first turtle was 5 seconds ago.
- What we really want to ask is, *"What was the pose of /turtle1 5 seconds ago, relative to the current position of the /turtle2?"*

RESULT:

EXPERIMENT:5

AIM: Building a Visual Robot Model with URDF from Scratch, Building a Movable RobotModel with URDF, Adding Physical and Collision Properties to a URDF Model.

Many of the coolest and most useful capabilities of ROS and its community involve things like collision checking and dynamic path planning. It's frequently useful to have a code-independent, human-readable way to describe the geometry of robots and their cells.The Unified Robot Description Format (URDF) is the most popular of these formats today.

The Unified Robotic Description Format (URDF) is an XML file format used in ROS to describe all elements of a robot.

Building a Visual Robot Model with URDF from Scratch

First, we're going to build one simple shape (cylinder).

Steps to be followed:

1. Create a workspace
   cd name of your workspace
2. Create a URDF folder to keep the urdf files
   mkdir urdf

3. Change directory into urdf folder

   cd urdf

   gedit myfirst.urdf

```
<?xml version="1.0"?>
<robot name="myfirst">
 <link name="base_link">
   <visual>
     <geometry>
       <cylinder length="0.6" radius="0.2"/>
     </geometry>
   </visual>
 </link>
</robot>
```

this is a robot with the name myfirst, that contains only one link (a.k.a. part), whose visual component is just a cylinder 0.6 meters long with a 0.2 meter radius.

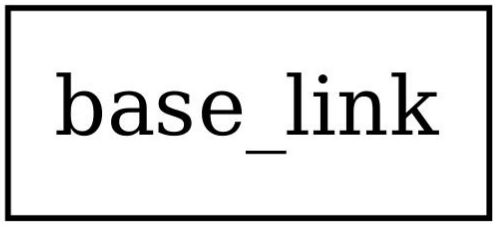
4. Now run the check command in the terminal

check_urdf myfirst.urdf

```
administrator@administrator-B365M-GAMING-HD:~/urdf_tut/urdf$ check_urdf shape.ur
df
robot name is: myfirst
---------- Successfully Parsed XML ---------------
root Link: base link has 0 child(ren)
```

5. Now visualize the URDF using graphiz:

urdf_to_graphiz myfirst.urdf

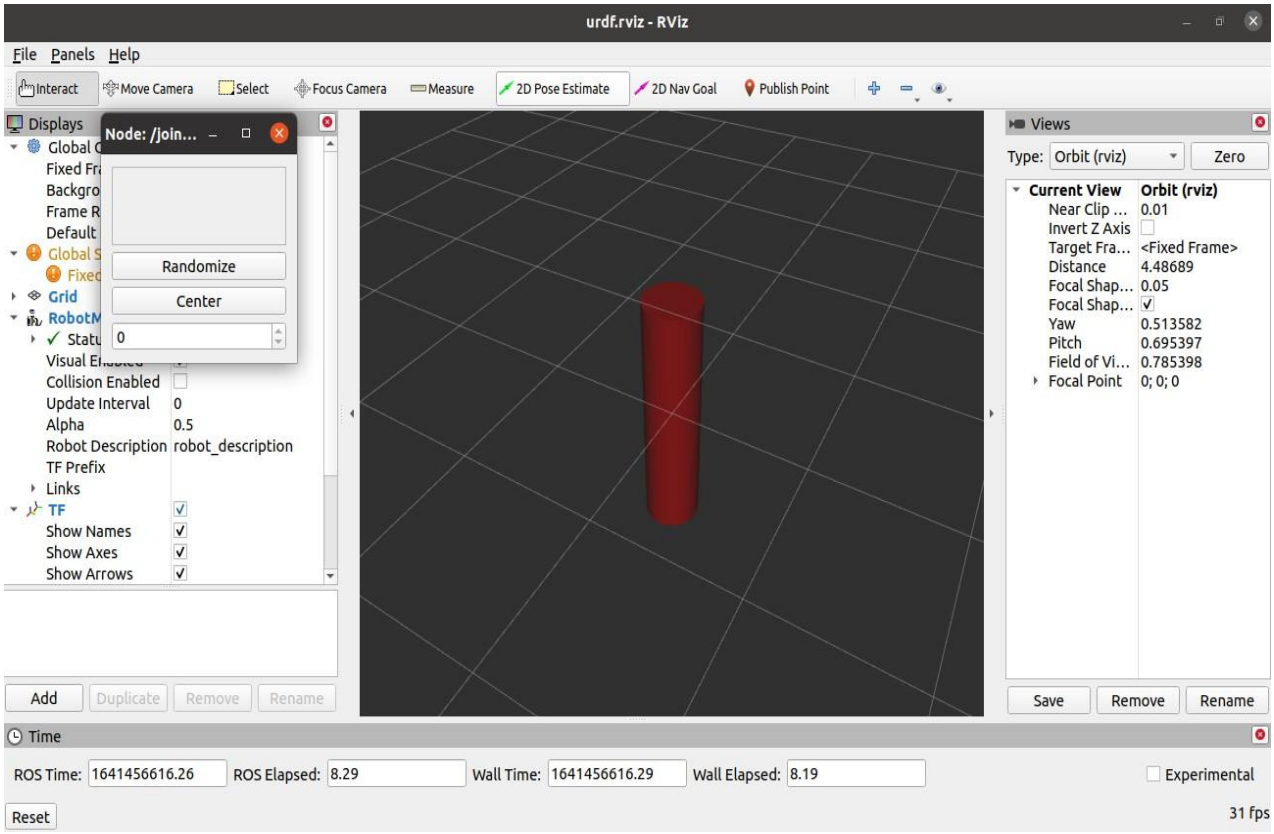


6. Change to workspace folder

cd name of your workspace
roslaunch urdf_tutorial display.launch model:=urdf/myfirst.urdf

This does three things. It

1. Loads the specified model into the parameter server
2. Runs nodes to publish sensor_msgs/JointState and transforms (more on these later)
3. Starts Rviz with a configuration file



Things to note:

The fixed frame is the transform frame where the center of the grid is located. Here, it's a frame defined by our one link, base_link.

The visual element (the cylinder) has its origin at the center of its geometry as a default. Hence, half the cylinder is below the grid.

Add multiple shapes/links

Now we are going to build multiple shapes. If we just add more link elements to the urdf, the parser won't know where to put them. So, we have to add joints. Joint elements can refer to both flexible and inflexible joints. We'll start with inflexible, or fixed joints

cd name of your workspace

cd urdf

gedit multipleshapes.urdf

```
<?xml version="1.0"?>
<robot name="multipleshapes">
```

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </visual>
</link>

<link name="right_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
  </visual>
</link>

<joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
</joint>

</robot>
```

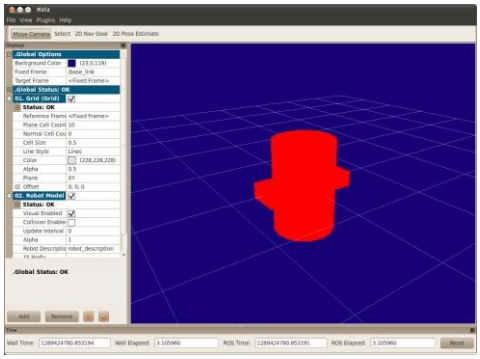
Note how we defined a 0.6m x 0.1m x 0.2m box

The joint is defined in terms of a parent and a child. URDF is ultimately a tree structure with one root link. This means that the leg's position is dependent on the base_link's position

check_urdf multipleshapes.urdf

urdf_to_graphiz multipleshapes.urdf

roslaunch urdf_tutorial display.launch model:=urdf/multipleshapes.urdf



Both of the shapes overlap with each other, because they share the same origin. If we want them not to overlap we must define more origins.

To define the origin of the joint

R2D2's leg attaches to the top half of his torso, on the side. So that's where we specify the origin of the JOINT to be. Also, it doesn't attach to the middle of the leg, it attaches to the upper part, so we must offset the origin for the leg as well. We also rotate the leg so it is upright

cd name of your workspace

cd urdf

gedit origins.urdf

```
<?xml version="1.0"?>
<robot name="origins">
 <link name="base_link">
  <visual>
   <geometry>
    <cylinder length="0.6" radius="0.2"/>
   </geometry>
  </visual>
 </link>

 <link name="right_leg">
  <visual>
   <geometry>
    <box size="0.6 0.1 0.2"/>
   </geometry>
   <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
  </visual>
 </link>

 <joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0 -0.22 0.25"/>
 </joint>

</robot>
```

- Let's start by examining the joint's origin. It is defined in terms of the parent's reference frame. So we are -0.22 meters in the y direction (to our left, but to the right relative to the axes) and 0.25 meters in the z direction (up). This means that the origin for the child link will be up and to the right, regardless of the child link's visual origin tag. Since we didn't specify a rpy (roll pitch yaw) attribute, the child frame will be default have the same orientation as the parent frame.

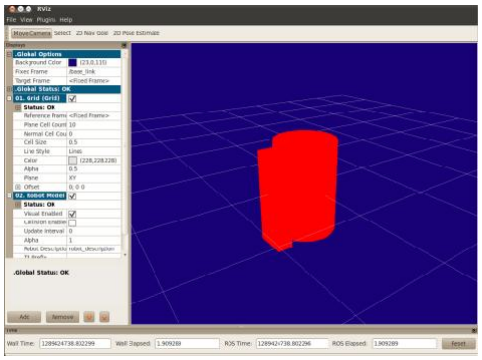
- Now, looking at the leg's visual origin, it has both a xyz and rpy offset. This defines where the center of the visual element should be, relative to its origin. Since we want the leg to attach at the top, we offset the origin down by setting the z offset to be -0.3 meters. And since we want the long part of the leg to be parallel to the z axis, we rotate the visual part PI/2 around the Y axis.

check_urdf origins.urdf

urdf_to_graphiz origins.urdf

cd ..

roslaunch urdf_tutorial display.launch model:=urdf/origins.urdf



Adding Material tag

cd name of your workspace

cd urdf

gedit materials.urdf

```
<?xml version="1.0"?>
<robot name="materials">

 <material name="blue">
   <color rgba="0 0 0.8 1"/>
 </material>

 <material name="white">
   <color rgba="1 1 1 1"/>
 </material>
```

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue"/>
  </visual>
</link>

<link name="right_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>

<joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0 -0.22 0.25"/>
</joint>

<link name="left_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>

<joint name="base_to_left_leg" type="fixed">
  <parent link="base_link"/>
  <child link="left_leg"/>
  <origin xyz="0 0.22 0.25"/>
</joint>

</robot>
```

The body is now blue. We've defined a new material called "blue", with the red, green, blue and alpha channels defined as 0,0,0.8 and 1 respectively. All of the values can be in the range [0,1]. This material is then referenced by the base_link's visual element. The white material is defined similarly

You could also define the material tag from within the visual element, and even reference it in other links. No one will even complain if you redefine it though.

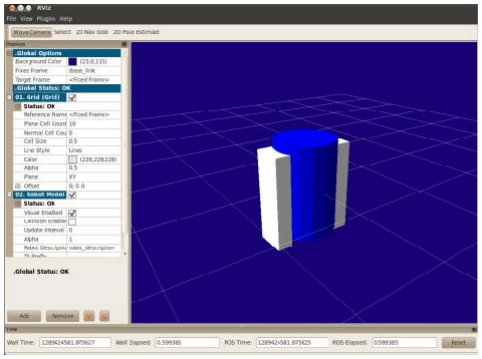
You can also use a texture to specify an image file to be used for coloring the object

check_urdf materials.urdf

urdf_to_graphiz materials.urdf

cd .. (should be in workspace)

roslaunch urdf_tutorial display.launch model:=urdf/materials.urdf



Finishing the Model

Now we finish the model off with a few more shapes: feet, wheels, and head.

cd name of your workspace

cd urdf

gedit visual.urdf

```
<?xml version="1.0"?>
<robot name="visual">

  <material name="blue">
    <color rgba="0 0 0.8 1"/>
  </material>
  <material name="black">
    <color rgba="0 0 0 1"/>
  </material>
  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>

  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
```

```
    </geometry>
    <material name="blue"/>
  </visual>
</link>

<link name="right_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>

<joint name="base_to_right_leg" type="fixed">
  <parent link="base_link"/>
  <child link="right_leg"/>
  <origin xyz="0 -0.22 0.25"/>
</joint>

<link name="right_base">
  <visual>
    <geometry>
      <box size="0.4 0.1 0.1"/>
    </geometry>
    <material name="white"/>
  </visual>
</link>

<joint name="right_base_joint" type="fixed">
  <parent link="right_leg"/>
  <child link="right_base"/>
  <origin xyz="0 0 -0.6"/>
</joint>

<link name="right_front_wheel">
  <visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>
<joint name="right_front_wheel_joint" type="fixed">
  <parent link="right_base"/>
  <child link="right_front_wheel"/>
  <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
```

```
</joint>

<link name="right_back_wheel">
  <visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>
<joint name="right_back_wheel_joint" type="fixed">
  <parent link="right_base"/>
  <child link="right_back_wheel"/>
  <origin rpy="0 0 0" xyz="-0.133333333333 0 -0.085"/>
</joint>

<link name="left_leg">
  <visual>
    <geometry>
      <box size="0.6 0.1 0.2"/>
    </geometry>
    <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
    <material name="white"/>
  </visual>
</link>

<joint name="base_to_left_leg" type="fixed">
  <parent link="base_link"/>
  <child link="left_leg"/>
  <origin xyz="0 0.22 0.25"/>
</joint>

<link name="left_base">
  <visual>
    <geometry>
      <box size="0.4 0.1 0.1"/>
    </geometry>
    <material name="white"/>
  </visual>
</link>

<joint name="left_base_joint" type="fixed">
  <parent link="left_leg"/>
  <child link="left_base"/>
  <origin xyz="0 0 -0.6"/>
</joint>

<link name="left_front_wheel">
```

```
  <visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>
<joint name="left_front_wheel_joint" type="fixed">
  <parent link="left_base"/>
  <child link="left_front_wheel"/>
  <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
</joint>

<link name="left_back_wheel">
  <visual>
    <origin rpy="1.57075 0 0" xyz="0 0 0"/>
    <geometry>
      <cylinder length="0.1" radius="0.035"/>
    </geometry>
    <material name="black"/>
  </visual>
</link>
<joint name="left_back_wheel_joint" type="fixed">
  <parent link="left_base"/>
  <child link="left_back_wheel"/>
  <origin rpy="0 0 0" xyz="-0.133333333333 0 -0.085"/>
</joint>

<joint name="gripper_extension" type="fixed">
  <parent link="base_link"/>
  <child link="gripper_pole"/>
  <origin rpy="0 0 0" xyz="0.19 0 0.2"/>
</joint>

<link name="gripper_pole">
  <visual>
    <geometry>
      <cylinder length="0.2" radius="0.01"/>
    </geometry>
    <origin rpy="0 1.57075 0 " xyz="0.1 0 0"/>
  </visual>
</link>

<joint name="left_gripper_joint" type="fixed">
  <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
  <parent link="gripper_pole"/>
  <child link="left_gripper"/>
</joint>
```

```
<link name="left_gripper">
  <visual>
    <origin rpy="0.0 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://urdf_tutorial/meshes/l_finger.dae"/>
    </geometry>
  </visual>
</link>

<joint name="left_tip_joint" type="fixed">
  <parent link="left_gripper"/>
  <child link="left_tip"/>
</joint>

<link name="left_tip">
  <visual>
    <origin rpy="0.0 0 0" xyz="0.09137 0.00495 0"/>
    <geometry>
      <mesh filename="package://urdf_tutorial/meshes/l_finger_tip.dae"/>
    </geometry>
  </visual>
</link>
<joint name="right_gripper_joint" type="fixed">
  <origin rpy="0 0 0" xyz="0.2 -0.01 0"/>
  <parent link="gripper_pole"/>
  <child link="right_gripper"/>
</joint>

<link name="right_gripper">
  <visual>
    <origin rpy="-3.1415 0 0" xyz="0 0 0"/>
    <geometry>
      <mesh filename="package://urdf_tutorial/meshes/l_finger.dae"/>
    </geometry>
  </visual>
</link>

<joint name="right_tip_joint" type="fixed">
  <parent link="right_gripper"/>
  <child link="right_tip"/>
</joint>

<link name="right_tip">
  <visual>
    <origin rpy="-3.1415 0 0" xyz="0.09137 0.00495 0"/>
    <geometry>
      <mesh filename="package://urdf_tutorial/meshes/l_finger_tip.dae"/>
    </geometry>
```

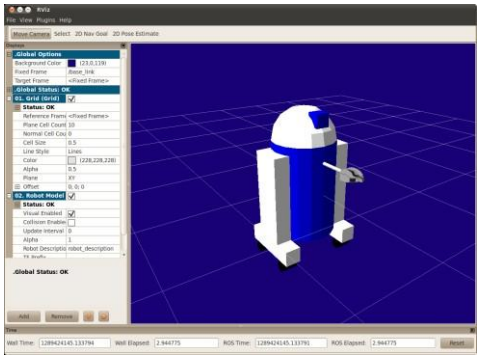
```
  </visual>
 </link>

 <link name="head">
  <visual>
   <geometry>
    <sphere radius="0.2"/>
   </geometry>
   <material name="white"/>
  </visual>
 </link>
 <joint name="head_swivel" type="fixed">
  <parent link="base_link"/>
  <child link="head"/>
  <origin xyz="0 0 0.3"/>
 </joint>

 <link name="box">
  <visual>
   <geometry>
    <box size="0.08 0.08 0.08"/>
   </geometry>
   <material name="blue"/>
  </visual>
 </link>

 <joint name="tobox" type="fixed">
  <parent link="head"/>
  <child link="box"/>
  <origin xyz="0.1814 0 0.1414"/>
 </joint>
</robot>
```

check_urdf visual.urdf

urdf_to_graphiz visual.urdf

cd ..

roslaunch urdf_tutorial display.launch model:=urdf/visual.urdf



Building a Movable RobotModel with URDF

To revise the R2D2 model we made so that it has movable joints. In the previous model, all of the joints were fixed. Now we'll explore three other important types of joints: continuous, revolute and prismatic.

cd name of your workspace

cd urdf

gedit flexible.urdf

```
  <?xml version="1.0"?>
<robot name="visual">

<material name="blue">
            <color rgba="0 0 0.8 1"/>
 </material>
 <material name="black">
            <color rgba="0 0 0 1"/>
 </material>
 <material name="white">
            <color rgba="1 1 1 1"/>
 </material>

 <link name="base_link">
            <visual>
            <geometry>
            <cylinder length="0.6" radius="0.2"/>
            </geometry>
            <material name="blue"/>
            </visual>
 </link>

 <link name="right_leg">
            <visual>
            <geometry>
            <box size="0.6 0.1 0.2"/>
```

```
        </geometry>
        <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
        <material name="white"/>
        </visual>
</link>

<joint name="base_to_right_leg" type="fixed">
        <parent link="base_link"/>
        <child link="right_leg"/>
        <origin xyz="0 -0.22 0.25"/>
</joint>

<link name="right_base">
        <visual>
        <geometry>
        <box size="0.4 0.1 0.1"/>
        </geometry>
        <material name="white"/>
        </visual>
</link>

<joint name="right_base_joint" type="fixed">
        <parent link="right_leg"/>
        <child link="right_base"/>
        <origin xyz="0 0 -0.6"/>
</joint>

<link name="right_front_wheel">
        <visual>
        <origin rpy="1.57075 0 0" xyz="0 0 0"/>
        <geometry>
        <cylinder length="0.1" radius="0.035"/>
        </geometry>
        <material name="black"/>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        </visual>
</link>
<joint name="right_front_wheel_joint" type="continuous">
        <axis rpy="0 0 0" xyz="0 1 0"/>
        <parent link="right_base"/>
        <child link="right_front_wheel"/>
        <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
</joint>

<link name="right_back_wheel">
        <visual>
        <origin rpy="1.57075 0 0" xyz="0 0 0"/>
        <geometry>
        <cylinder length="0.1" radius="0.035"/>
```

```
            </geometry>
            <material name="black"/>
            </visual>
</link>
<joint name="right_back_wheel_joint" type="continuous">
            <axis rpy="0 0 0" xyz="0 1 0"/>
            <parent link="right_base"/>
            <child link="right_back_wheel"/>
            <origin rpy="0 0 0" xyz="-0.133333333333 0 -0.085"/>
</joint>

<link name="left_leg">
            <visual>
            <geometry>
            <box size="0.6 0.1 0.2"/>
            </geometry>
            <origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
            <material name="white"/>
            </visual>
</link>

<joint name="base_to_left_leg" type="fixed">
            <parent link="base_link"/>
            <child link="left_leg"/>
            <origin xyz="0 0.22 0.25"/>
</joint>

<link name="left_base">
            <visual>
            <geometry>
            <box size="0.4 0.1 0.1"/>
            </geometry>
            <material name="white"/>
            </visual>
</link>

<joint name="left_base_joint" type="fixed">
            <parent link="left_leg"/>
            <child link="left_base"/>
            <origin xyz="0 0 -0.6"/>
</joint>

<link name="left_front_wheel">
            <visual>
            <origin rpy="1.57075 0 0" xyz="0 0 0"/>
            <geometry>
            <cylinder length="0.1" radius="0.035"/>
            </geometry>
            <material name="black"/>
```

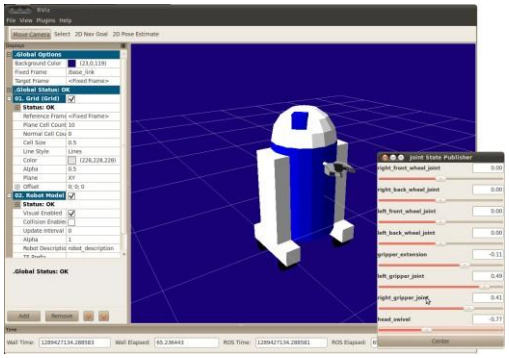
```
        </visual>
</link>
<joint name="left_front_wheel_joint" type="continuous">
        <axis rpy="0 0 0" xyz="0 1 0"/>
        <parent link="left_base"/>
        <child link="left_front_wheel"/>
        <origin rpy="0 0 0" xyz="0.133333333333 0 -0.085"/>
</joint>

<link name="left_back_wheel">
        <visual>
        <origin rpy="1.57075 0 0" xyz="0 0 0"/>
        <geometry>
        <cylinder length="0.1" radius="0.035"/>
        </geometry>
        <material name="black"/>
        </visual>
</link>
<joint name="left_back_wheel_joint" type="continuous">
        <axis rpy="0 0 0" xyz="0 1 0"/>
        <parent link="left_base"/>
        <child link="left_back_wheel"/>
        <origin rpy="0 0 0" xyz="-0.133333333333 0 -0.085"/>
</joint>

<joint name="gripper_extension" type="prismatic">
        <parent link="base_link"/>
        <child link="gripper_pole"/>
        <limit effort="1000.0" lower="-0.38" upper="0" velocity="0.5"/>
        <origin rpy="0 0 0" xyz="0.19 0 0.2"/>
</joint>

<link name="gripper_pole">
        <visual>
        <geometry>
        <cylinder length="0.2" radius="0.01"/>
        </geometry>
        <origin rpy="0 1.57075 0 " xyz="0.1 0 0"/>
        </visual>
</link>

<joint name="left_gripper_joint" type="revolute">
        <axis xyz="0 0 1"/>
        <limit effort="1000.0" lower="0.0" upper="0.548" velocity="0.5"/>
        <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
        <parent link="gripper_pole"/>
        <child link="left_gripper"/>
</joint>
```

```
<link name="left_gripper">
        <visual>
        <origin rpy="0.0 0 0" xyz="0 0 0"/>
        <geometry>
        <mesh filename="package://urdf_tutorial/meshes/l_finger.dae"/>
        </geometry>
        </visual>
</link>

<joint name="left_tip_joint" type="fixed">
        <parent link="left_gripper"/>
        <child link="left_tip"/>
</joint>

<link name="left_tip">
        <visual>
        <origin rpy="0.0 0 0" xyz="0.09137 0.00495 0"/>
        <geometry>
        <mesh filename="package://urdf_tutorial/meshes/l_finger_tip.dae"/>
        </geometry>
        </visual>
</link>
<joint name="right_gripper_joint" type="revolute">
        <axis xyz="0 0 -1"/>
        <limit effort="1000.0" lower="0.0" upper="0.548" velocity="0.5"/>
        <origin rpy="0 0 0" xyz="0.2 -0.01 0"/>
        <parent link="gripper_pole"/>
        <child link="right_gripper"/>
</joint>

<link name="right_gripper">
        <visual>
        <origin rpy="-3.1415 0 0" xyz="0 0 0"/>
        <geometry>
        <mesh filename="package://urdf_tutorial/meshes/l_finger.dae"/>
        </geometry>
        </visual>
</link>

<joint name="right_tip_joint" type="fixed">
        <parent link="right_gripper"/>
        <child link="right_tip"/>
</joint>

<link name="right_tip">
        <visual>
        <origin rpy="-3.1415 0 0" xyz="0.09137 0.00495 0"/>
        <geometry>
        <mesh filename="package://urdf_tutorial/meshes/l_finger_tip.dae"/>
```

```
            </geometry>
            </visual>
 </link>

 <link name="head">
            <visual>
            <geometry>
            <sphere radius="0.2"/>
            </geometry>
            <material name="white"/>
            </visual>
 </link>
 <joint name="head_swivel" type="continuous">
            <parent link="base_link"/>
            <child link="head"/>
            <axis xyz="0 0 1"/>
            <origin xyz="0 0 0.3"/>
 </joint>

 <link name="box">
            <visual>
            <geometry>
            <box size="0.08 0.08 0.08"/>
            </geometry>
            <material name="blue"/>
            </visual>
 </link>

 <joint name="tobox" type="fixed">
            <parent link="head"/>
            <child link="box"/>
            <origin xyz="0.1814 0 0.1414"/>
 </joint>
</robot>
```

cd ..

roslaunch urdf_tutorial display.launch model:=urdf/flexible.urdf



This will also pop up a GUI that allows you to control the values of all the non-fixed joints.

Adding Physical and Collision Properties to a URDF Model

Collision

In order to get collision detection to work or to simulate the robot in something like Gazebo, we need to define a collision element as well.

The collision element is a direct subelement of the link object, at the same level as the visual tag.

```
<link name="base_link">
   <visual>
     <geometry>
       <cylinder length="0.6" radius="0.2"/>
     </geometry>
     <material name="blue">
       <color rgba="0 0 .8 1"/>
     </material>
   </visual>
   <collision>
     <geometry>
       <cylinder length="0.6" radius="0.2"/>
     </geometry>
   </collision>
 </link>
```

The collision element defines its shape the same way the visual element does, with a geometry tag. The format for the geometry tag is exactly the same here as with the visual.

You can also specify an origin in the same way as a subelement of the collision tag (as with the visual).

Physical Properties

Inertia

Every link element being simulated needs an inertial tag.

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue">
      <color rgba="0 0 .8 1"/>
```

```
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4" iyz="0.0" izz="0.2"/>
  </inertial>
</link>
```

This element is also a subelement of the link object.

The mass is defined in kilograms.

The 3x3 rotational inertia matrix is specified with the inertia element. Since this is symmetrical, it can be represented by only 6 elements, as such.

ixx ixy ixz

ixy iyy iyzq

ixz iyz izz

RESULT:

Experiment.6

AIM: Familiarization with Gazebo--How to get Gazebo up and running, Creating and Spawning Custom URDF Objects in Simulation

What is Gazebo?

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

Typical uses of Gazebo include:

- testing robotics algorithms,
- designing robots,
- performing regression testing with realistic scenarios

A few key features of Gazebo include:

- multiple physics engines,
- a rich library of robot models and environments,
- a wide variety of sensors,
- convenient programmatic and graphical interfaces

World Files

The world description file contains all the elements in a simulation, including robots, lights, sensors, and static objects. This file is formatted using SDF (Simulation Description Format), and typically has a .world extension.

Gazebo runs two processes:

Server: Runs the physics loop and generates sensor data.

Libraries: Physics, Sensors, Rendering, Transport

Client: Provides user interaction and visualization of a simulation.

Libraries: Transport, Rendering, GUI

Elements within Simulation:

World

Collection of models, lights, plugins and global properties

Models

Collection of links, joints, sensors, and plugins

Links

Collection of collision and visual objects

Collision Objects

Geometry that defines a colliding surface

Visual Objects

Geometry that defines visual representation

Joints

Constraints between links

Sensors

Collect, process, and output data

Plugins

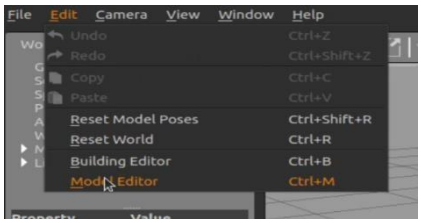
Code attached to a World, Model, Sensor, or the simulator itself

How to create a simple mobile base with the model editor.

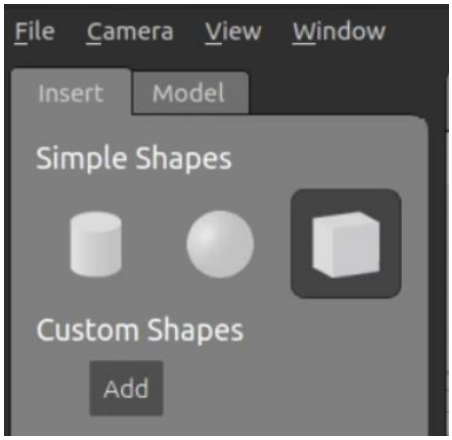
1. Type “gazebo” in the terminal

This will launch gazebo

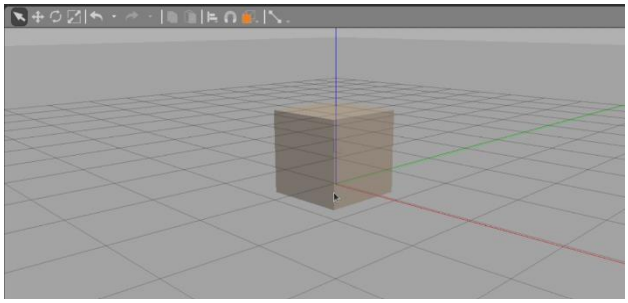
2. Open Model Editor



3.Select cube from simple shapes options

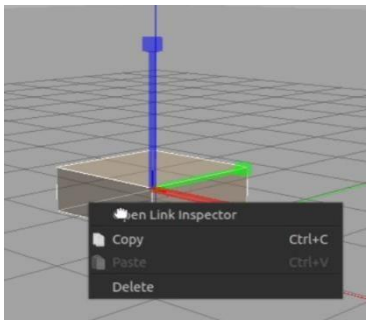


4.Place cube in the environment



Tips: Use short key “s” for scaling and short key “t “ for translation.

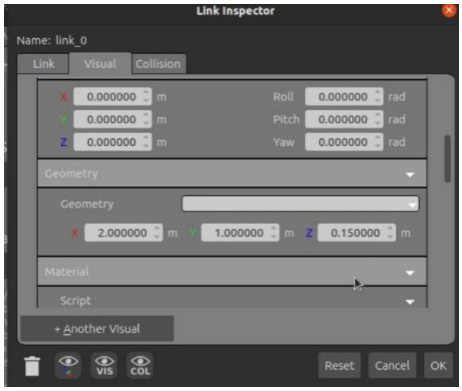
5. Open link inspector by selecting the object and right click.



6. Change the height of the robot base to 0.4m from the ground by changing the pose z value to 0.4m

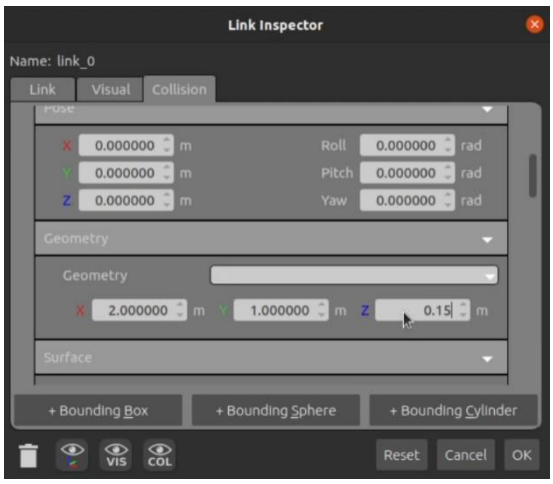
7.Select the **Visual** tab in the link inspector.

Enter x=2m;y=1m;z=0.15m corresponding to the Length width and height of the robot base in the Geometry field.

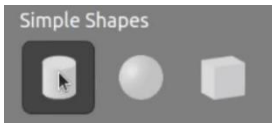


8. Select the **Collision** tab in the link inspector.

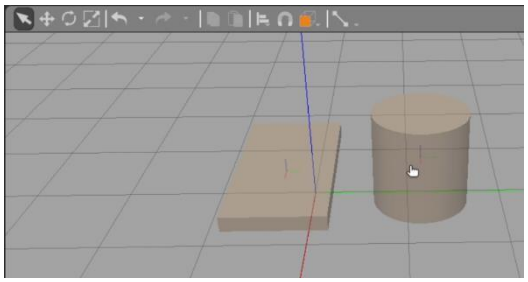
Enter x=2m;y=1m;z=0.15m corresponding to the Length width and height of the robot base in the Geometry field.



9. Select cylinder from simple shapes options



10. Place cylinder in the environment.



11. Open the link inspector by selecting the cylinder and right-click.

In order to make the cylinder a wheel, we have to rotate it 90 degrees along x-axis(Roll).

Enter Roll = 1.507 rad in pose option (1.5707 rad = 90 degree)

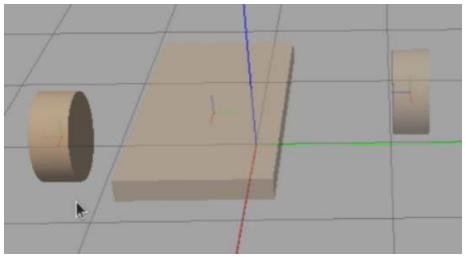
12. Select the **Visual** tab in the link inspector.

Enter Radius=0.3m and Length= 0.25m corresponding to radius and length of the robot base in the geometry field.

13. Select the **Collision** tab in the link inspector.

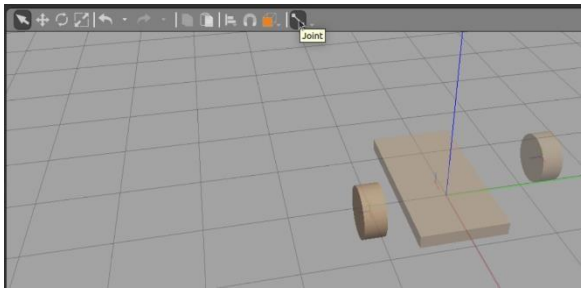
Enter Radius=0.3m and Length= 0.25m corresponding to radius and length of the robot base in the geometry field.

14. Copy and paste the cylinder for the second wheel.



15. To define the joints of the wheel.

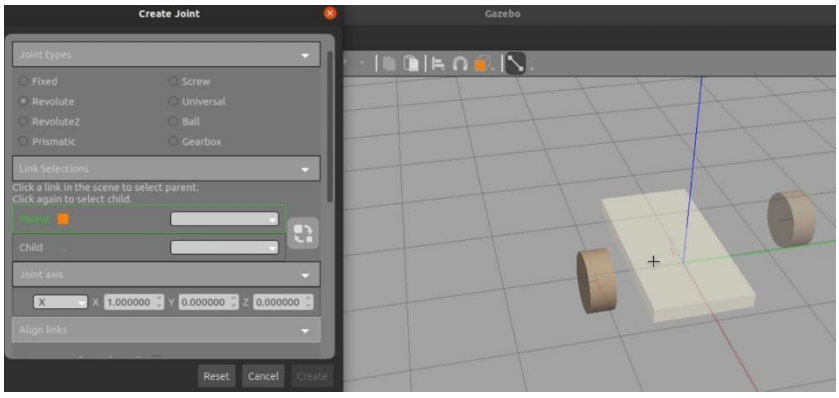
Select the joint option from the model editor tab.



16. Select revolute as the joint type.

Select robot base as the parent link by selecting the robot base using mouse pointer.

Select wheel 1 as the child link by selecting the wheel 1 using mouse pointer.

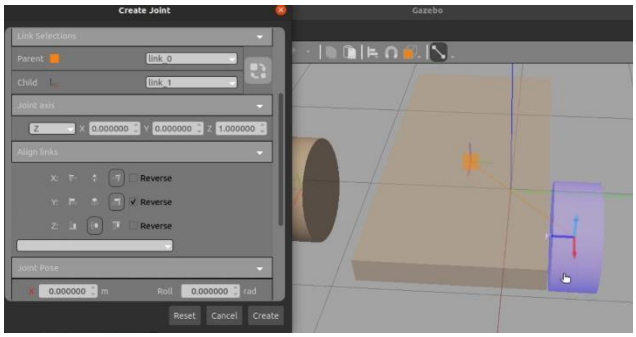


17. Select z-axis of the wheel as the rotation axis by selecting the Joint axis = “z”.

18. Place wheel 1 in the proper location of the robot base by the following selection.

In the align links field

Select x= max; y= max and reverse; z= center



19. To define the joints of the wheel 2.

Select the joint option from the tools.

Select revolute as the joint type.

Select robot base as the parent link by selecting the robot base using mouse pointer.

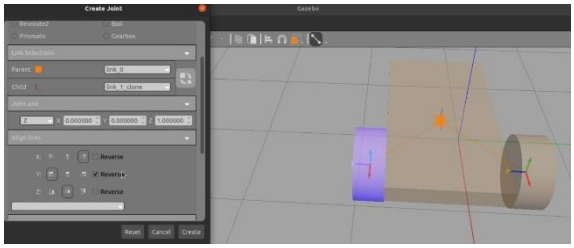
Select wheel 2 as the child link by selecting the wheel 2 using mouse pointer.

Select z-axis of the wheel as the rotation axis by selecting the Joint axis = “z”.

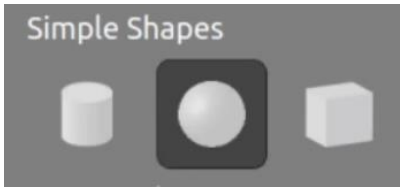
Place wheel 2 in the proper location of the robot base by the following selection.

In the align links field

Select x= max; y= min and reverse; z= center



20. Select sphere from the simple shape options and place it on the environment for creating the ball caster wheel.



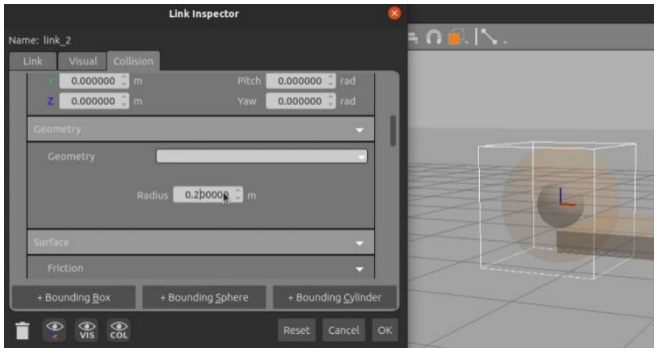
21. Open link inspector by selecting the sphere and right-click.

Select **Visual** tab in the link inspector.

Enter Radius=0.2m corresponding to the radius of the caster wheel in the geometry field.

Select **Collision** tab in the link inspector.

Enter Radius=0.2m corresponding to the radius of the caster wheel in the geometry field.



22. To define the joints of the castor wheel.

Select the joint option from the tools.

Select ball as the joint type.

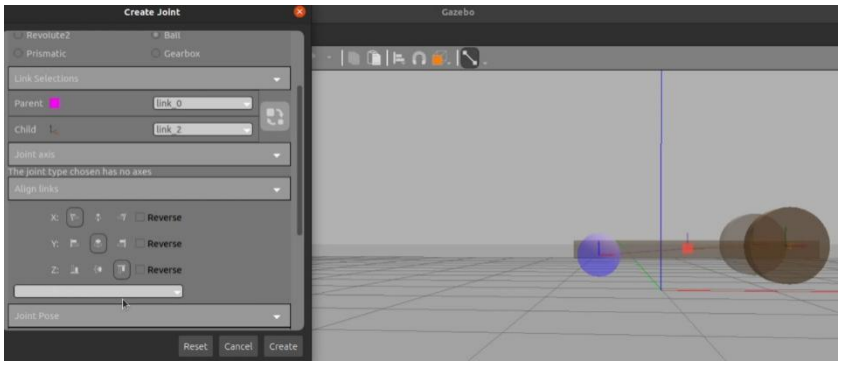
Select robot base as the parent link by selecting the robot base using mouse pointer.

Select castor wheel as the child link by selecting the sphere using mouse pointer.

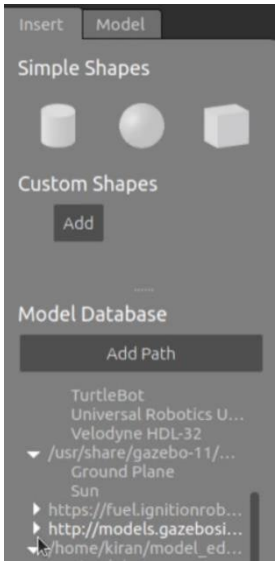
Place the castor wheel in the proper location of the robot base by the following selection.

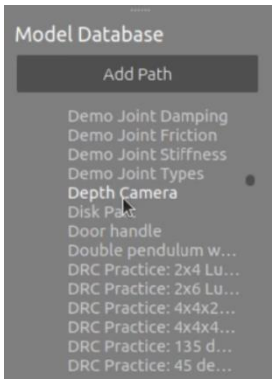
In the align links field

Select x= min; y= center; z= max

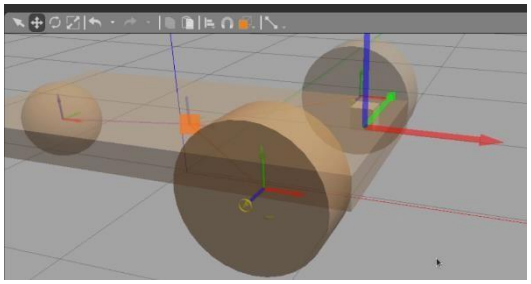


23. Place the Depth sensing camera by selecting Depth camera from the Model Database>> http:// models.gazebosim.





24. Place the depth camera on the robot by manually adjusting.



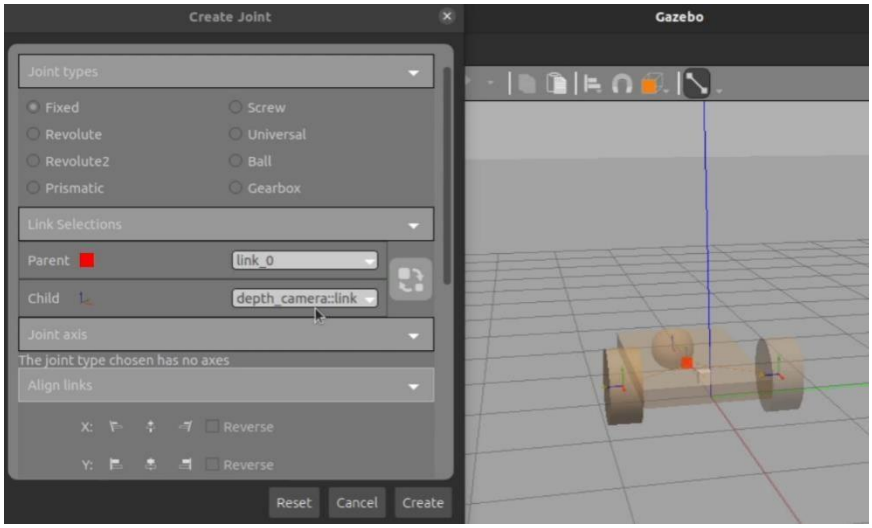
25. To define the joints of the Depth camera.

Select the joint option from the tools.

Select fixed as the joint type.

Select robot base as the parent link by selecting the robot base using mouse pointer.

Select Depth camera as the child link by selecting the depth camera using mouse pointer.

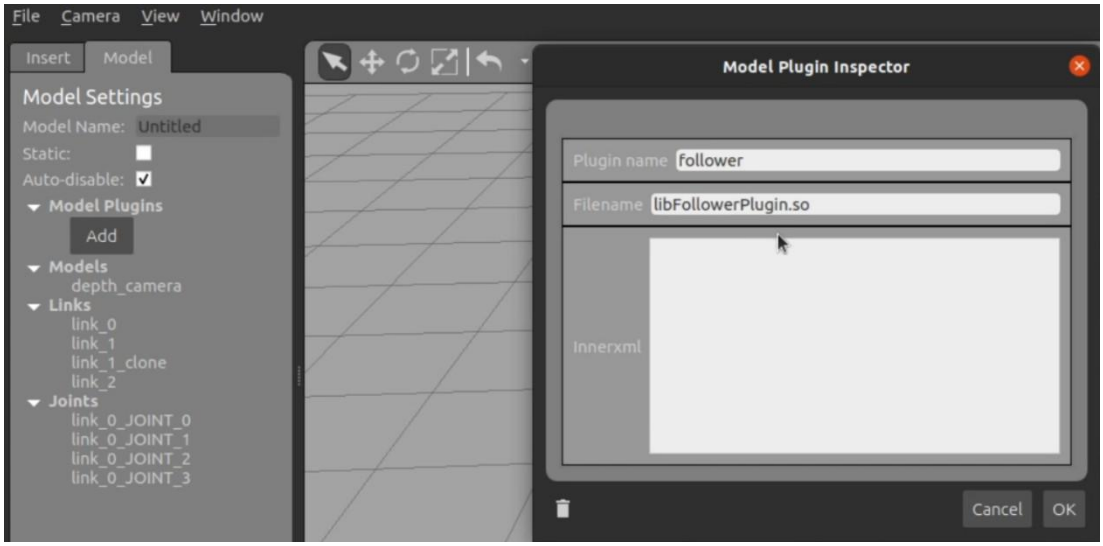


26. Add model plugin

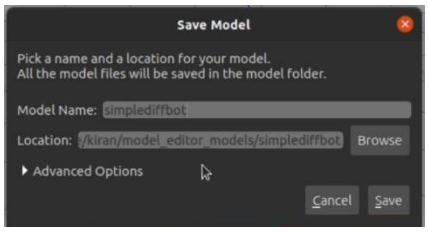
Enter the following

Plugin name= follower

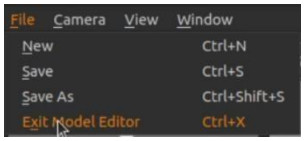
Filename = libFollowerPlugin.so



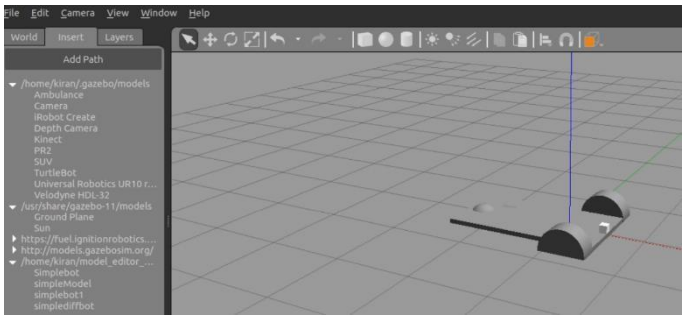
27. Save the model as “simplediffbot”



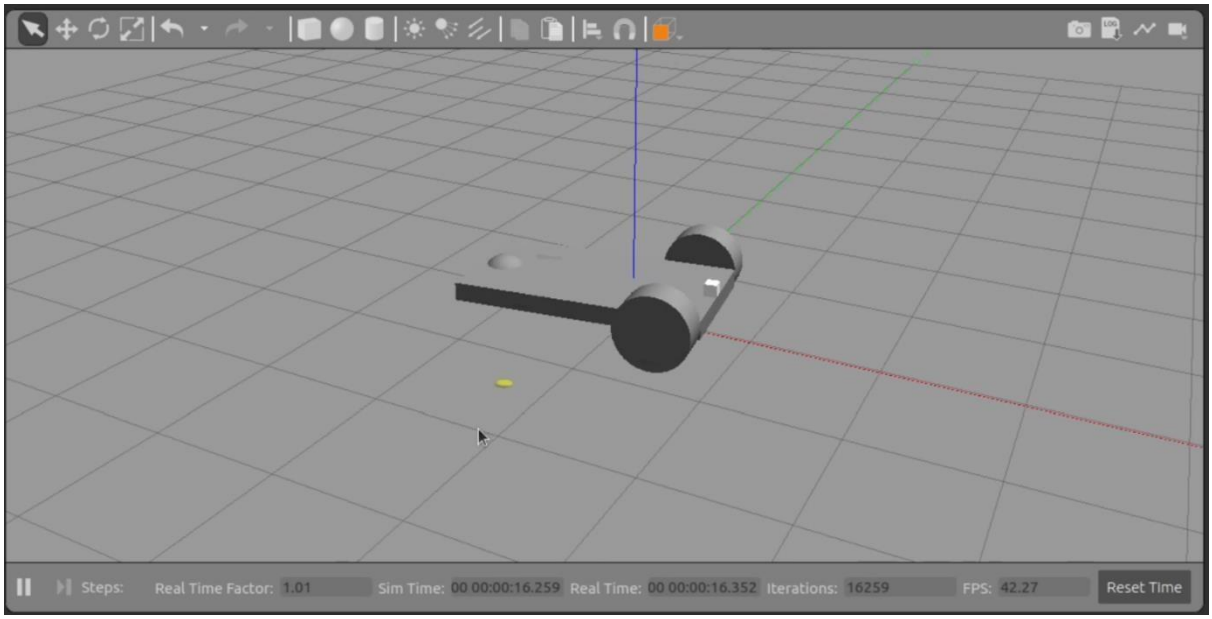
28. Exit model editor



29. Place the new model “simplediffbot” in the Gazebo environment.



30. Run the simulator using the play button at the bottom of the simulator. The robot will follow the obstacle in front using the depth camera.


Steps:
Real Time Factor: 1.01
Sim Time: 00 00:00:16.259
Real Time: 00 00:00:16.352
Iterations: 16259
FPS: 42.27
Reset Time

EXPERIMENT:7

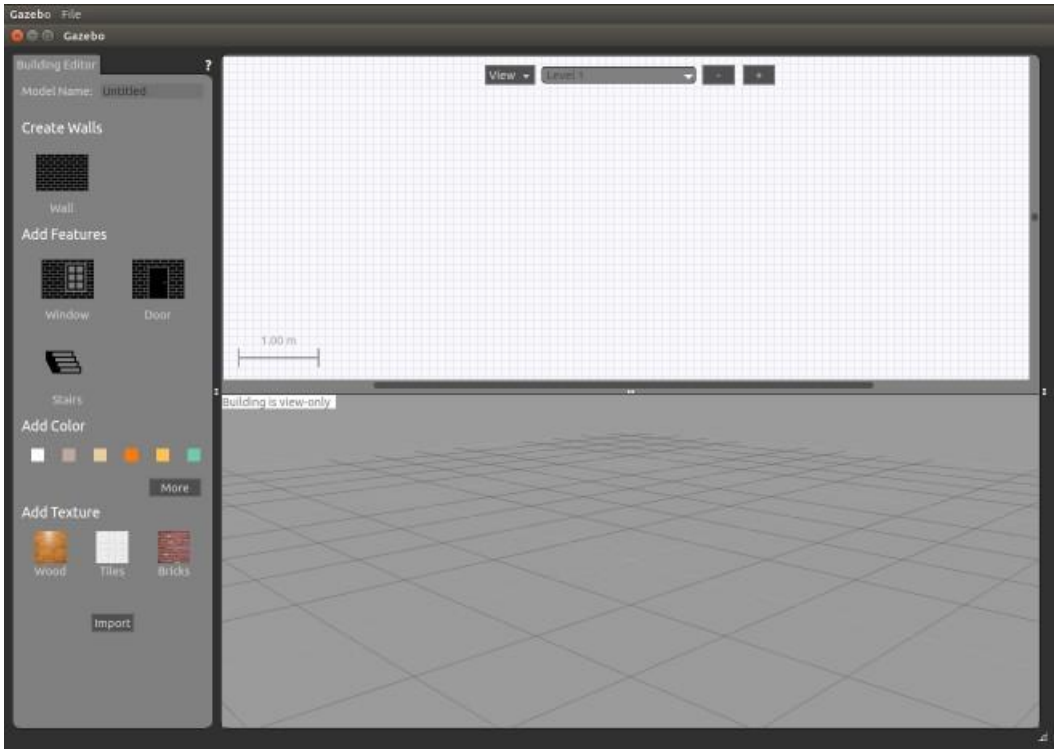
AIM: Create a Gazebo Custom World (Building Editor, Gazebo 3D Models), Add Sensor plugins like Laser, Kinect, etc. to URDF of mobile robot

7.1 Create a Gazebo Custom World (Building Editor, Gazebo 3D Models)

Start up gazebo.

gazebo

On the Edit menu, go to Building Editor, or hit Ctrl+B to open the editor.

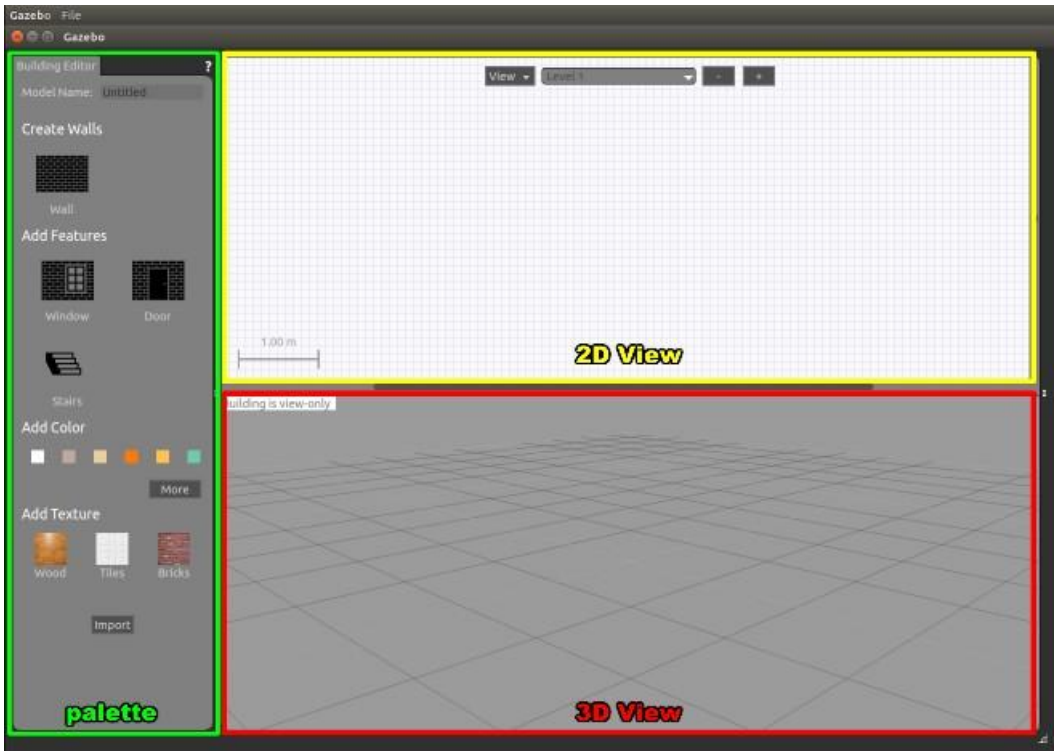


Graphical user interface

The editor is composed of the following 3 areas:

- The Palette, where you can choose features and materials for your building.
- The 2D View, where you can import a floor plan to trace over (optional) and insert walls, windows, doors and stairs.

- The 3D View, where you can see a preview of your building. It is also where you can assign colors and textures to different parts of your building.

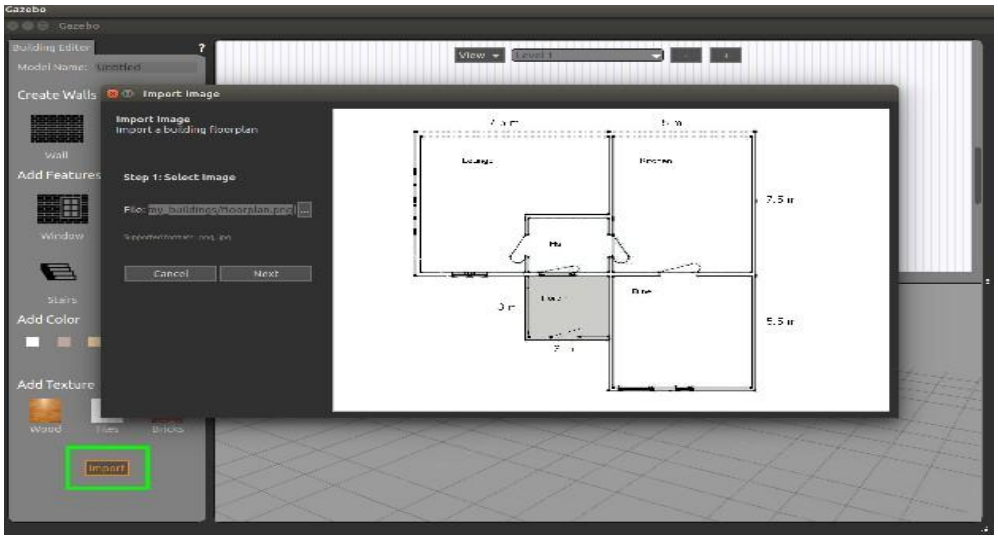


Import a floor plan

You may create a scene from scratch, or use an existing image as a template to trace over. This image can be, for example, a 2D laser scan of a building.

Click on the Import button. The Import Image dialog will come up.

Step 1: Choose the image you previously saved on your computer and click Next.



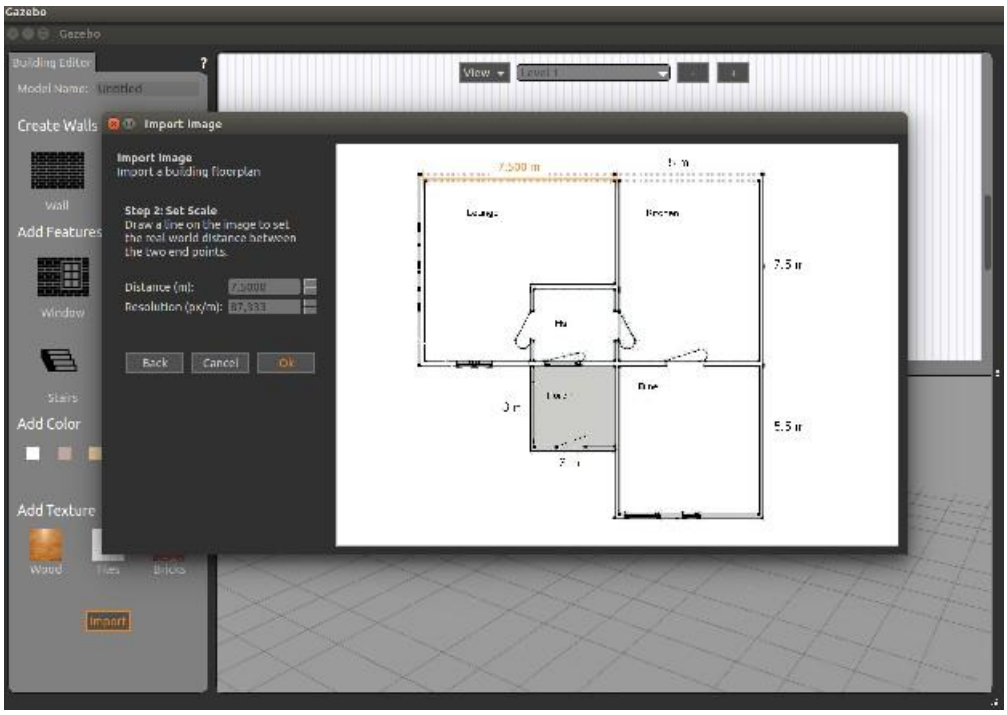
Step 2: To make sure the walls you trace over the image come up in the correct scale, you must set the image's resolution in pixels per meter (px/m). If we knew the resolution, we could directly type it in the dialog and click Ok. In this example we don't know the resolution, but we know the real-world distance between two points in the image (for example, the top wall of 7.5 m), so we can use that to calculate the resolution:

a. Click/release on one end of the wall. As you move the mouse, an orange line will appear as shown below.

b. Click/release at the end of the wall to complete the line.

c. Now type the distance in meters in the dialog (7.5 m in this case). The resolution will be automatically calculated for you based on the line you drew.

d. You can then click Ok.



The image will appear on the 2D View properly scaled.

Tip: Once you've added more levels, you can import a floor plan for each by repeating the same process.

Add features

Add walls

Trace all walls on the floor plan as follows. Keep in mind that we will attach windows and doors to the walls later, so here you can draw the walls over them. Don't worry too much if the walls are not perfect, we will edit them later.

On the Palette, click on Wall.

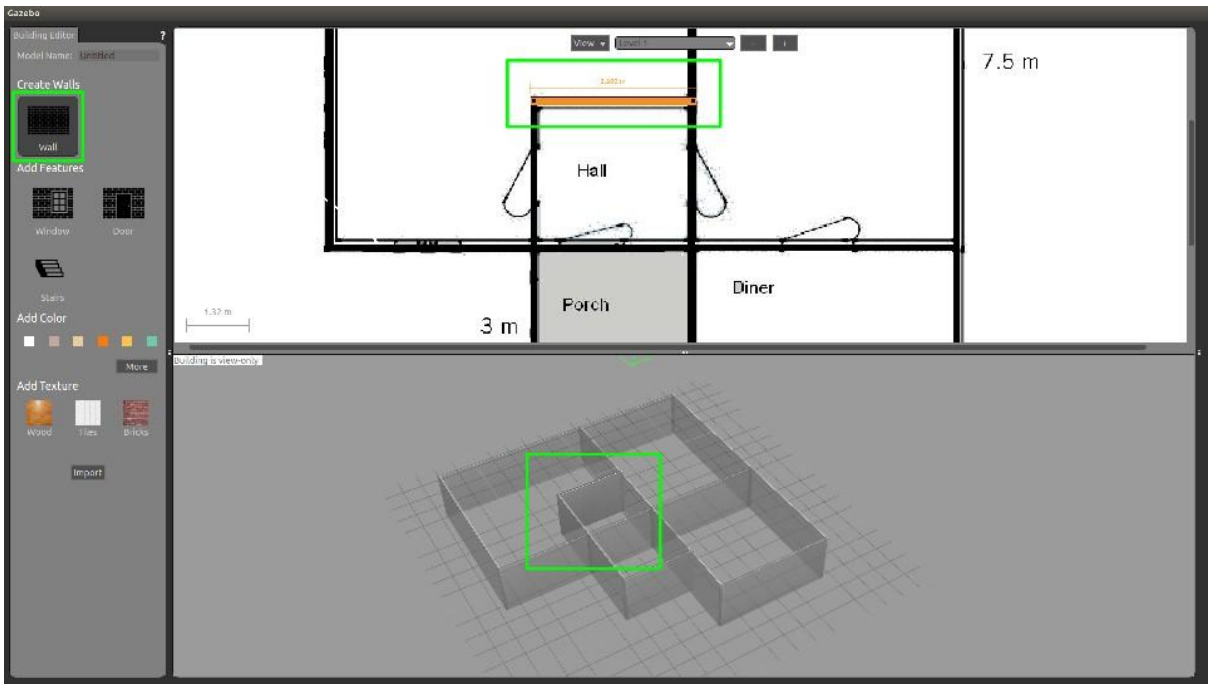
On the 2D View, click/release anywhere to start the wall. As you move the mouse, the wall's length is displayed.

Click again to end the current wall and start an adjacent wall.

Double-click to finish a wall without starting a new one.

Tip: You can right-click or press Esc to cancel drawing the current wall segment.

Tip: By default, walls snap to 15° and 0.25 m increments and also to the end points of existing walls. To override this, hold Shift while drawing.



Add windows and doors

Note: Currently, windows and doors are simple holes in the wall.

Let's insert windows and doors at the locations shown on the floor plan.

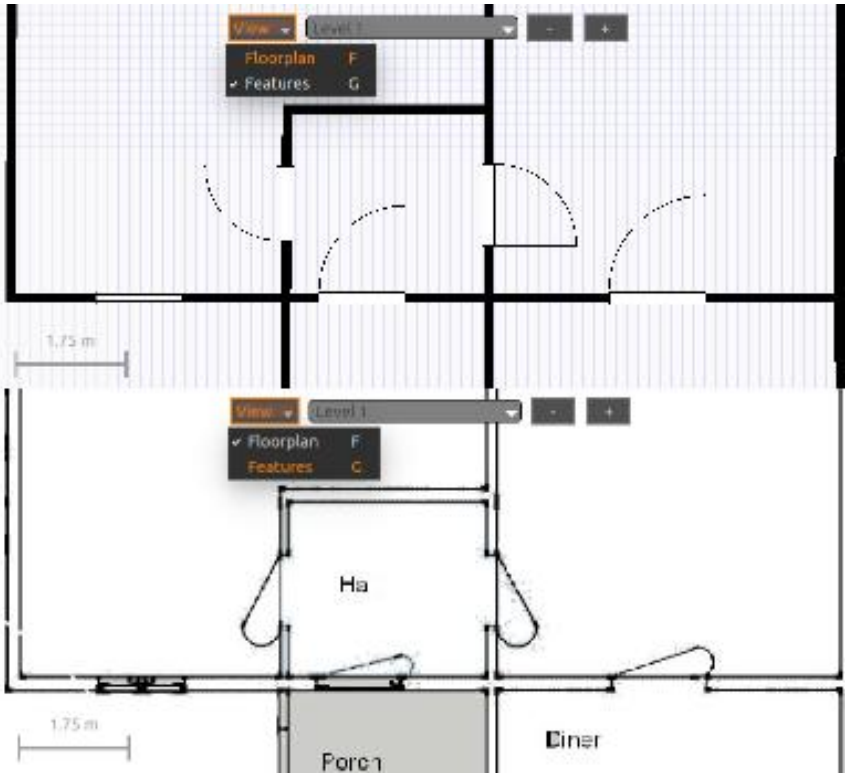
On the Palette, click on Window or Door.

As you move the mouse in the 2D view, the feature to be inserted moves with it, as does its counterpart in the 3D View.

Tip: Windows and doors automatically snap to walls as you hover over them. The distances to the ends of the wall are displayed as you move.

Click on the desired position to place the feature.

Tip: It might be difficult to see where the features are on your floor plan after the walls have been drawn on top of it. To make it easier, at the top of the 2D View, you can choose to view or hide the floor plan or features for the current level. You can also use hotkeys to toggle visibility, F for floor plan and G for features.



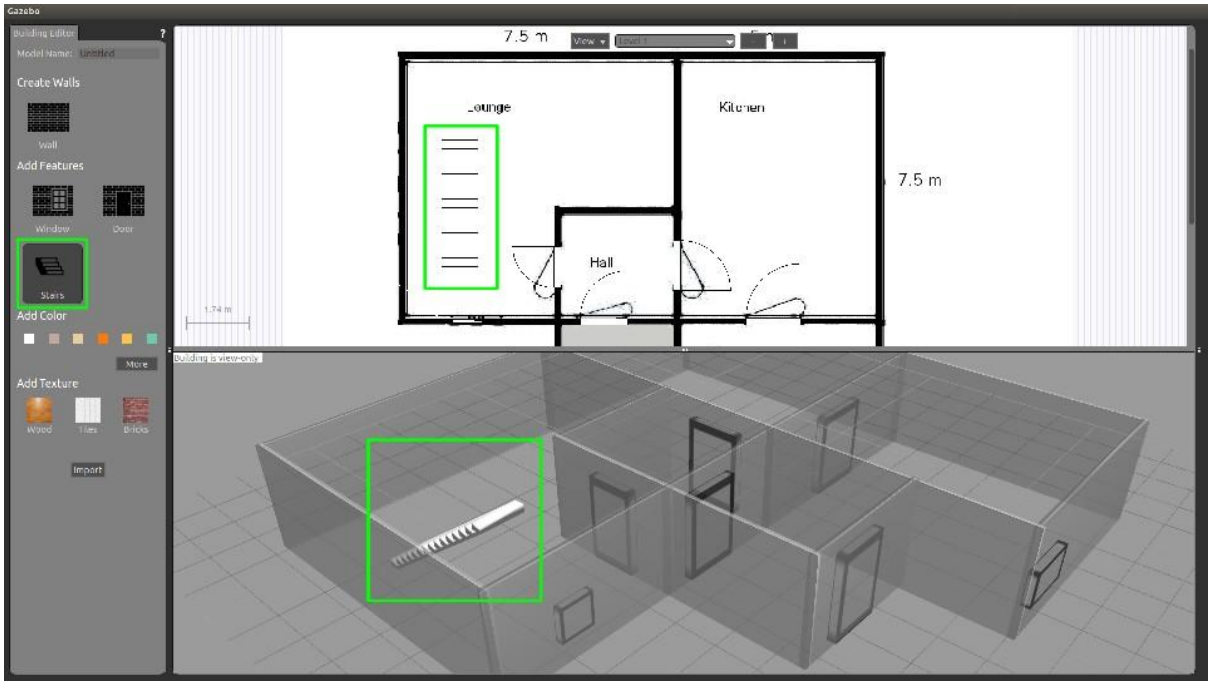
Add stairs

There are no staircases on this floor plan, but we will insert one anyways.

On the Palette, click on Stairs.

As you move the mouse in the 2D view, the staircase to be inserted moves with it, as does its counterpart in the 3D View.

Choose a position for your staircase and click to place it.



Add levels

We're pretty much done with Level 1. Let's add another level to our building so our staircase ends up somewhere.

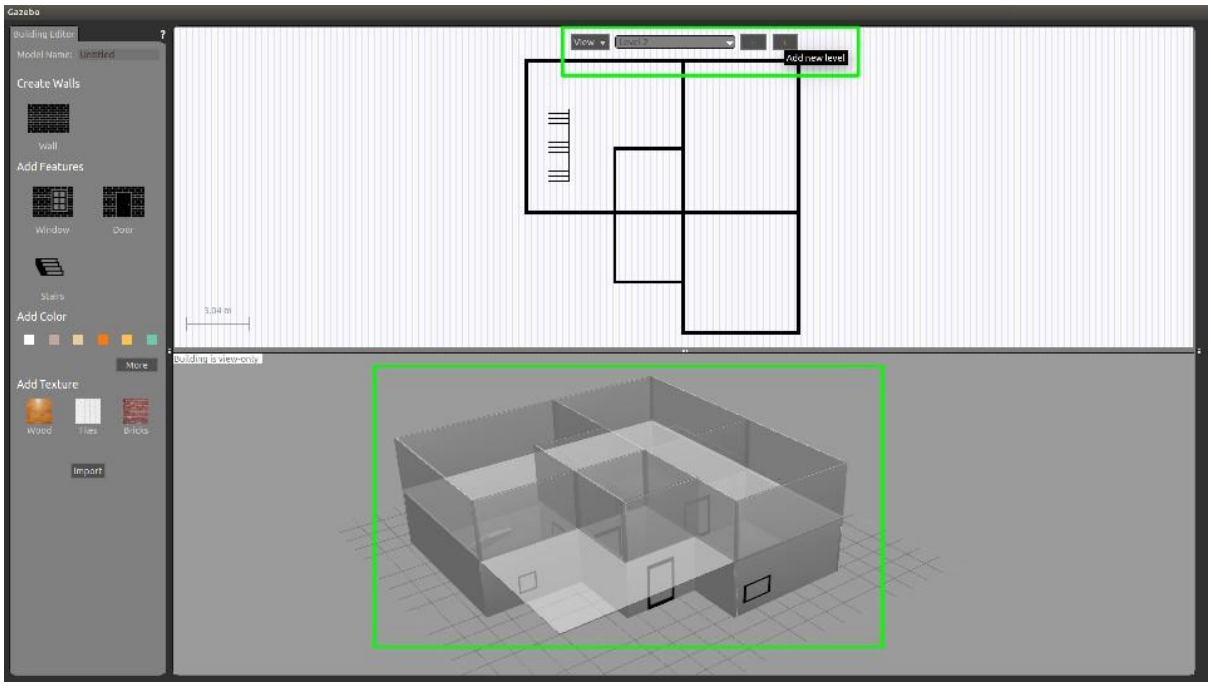
At the top of the 2D View, click on + to add a level. Alternatively, right-click the 2D View and choose Add a level.

When a new level is added, a floor is automatically inserted. If there are stairs on the level below, a hole above the stairs will be cut out from the floor when the building is saved.

Note: Currently, all floors are rectangular.

Tip: Before adding a level, make sure you have walls on the current level to build on top of.

Tip: Currently, all the walls from the level below are copied to the new level, with default materials. No other features are copied. You can manually delete the walls you don't want.



Edit your building

Note: Be careful when editing your building; the editor currently has no option to undo your actions.

Tip: All measurements are in meters.

Change levels

Since we added a level, we were brought to the new level in the 2D view. You can go back to Level 1 by choosing it from the drop-down list at the top of the 2D View.

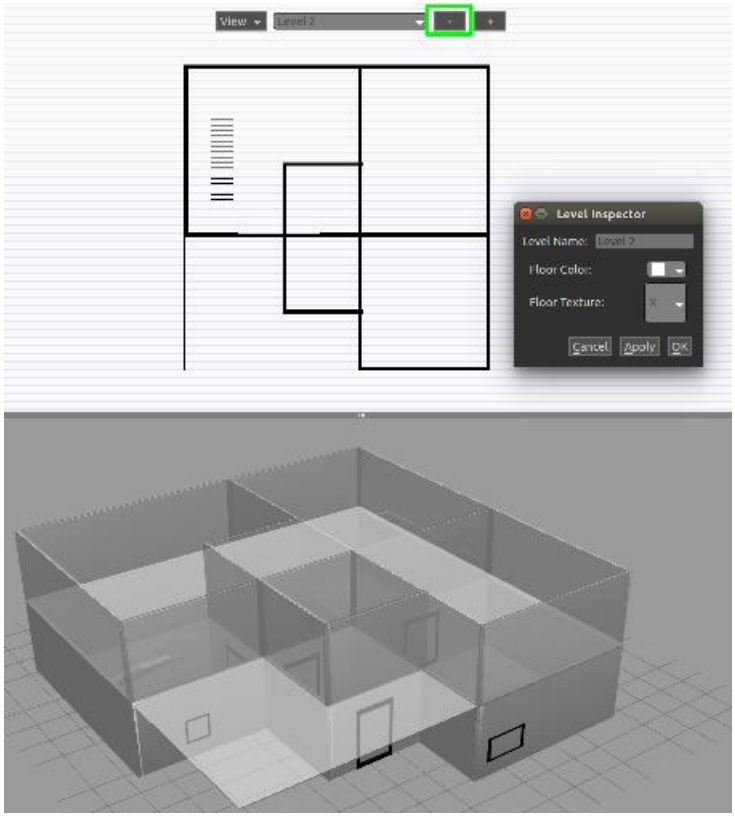
Tip: The level currently selected in the 2D View will appear as semi-transparent in the 3D View and all levels below it will appear opaque. Levels above will be hidden - but keep in mind they are still part of your building!

We can also edit some level configurations if we want.

Double-click the 2D View to open an inspector with level configuration options. Alternatively, right-click and choose Open Level Inspector.

You may have added levels that you don't want, or perhaps made a mess in the current level and would like to start it over.

To delete the current level, either press the - button at the top of the 2D View, or right-click and choose Delete Level.



Edit walls

We drew a lot of walls earlier, but maybe they didn't turn out exactly the way we wanted.

In the 2D View, click on the wall to be edited.

a. Translate the wall by dragging it to a new position.

b. Resize or rotate the wall by dragging one of its end points.

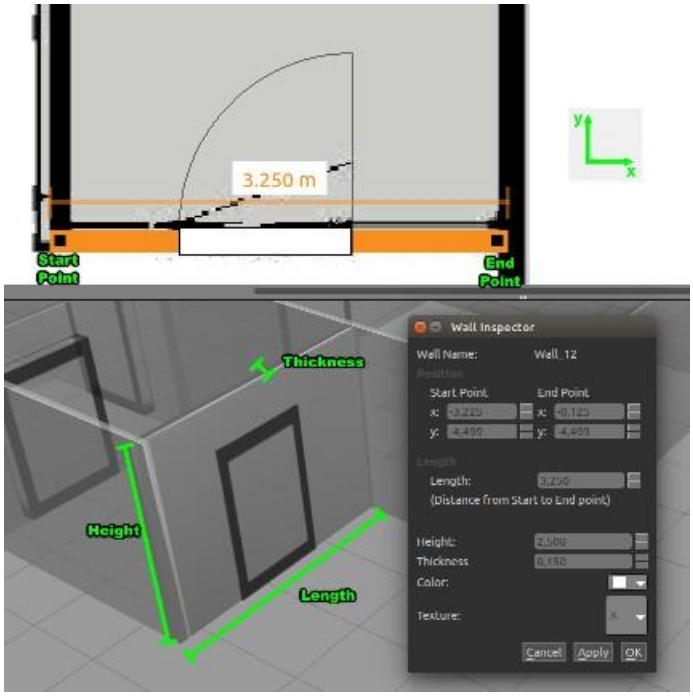
Tip: By default, walls snap to 15° and 0.25 m increments. To override this, hold Shift while drawing.

Double-click a wall in the 2D View to open an inspector with configuration options. Alternatively, right-click and choose Open Wall Inspector. Edit some fields and press Apply to preview the changes.

To delete a wall, either press the Delete key while it is selected, or right-click the wall in the 2D View and choose Delete.

Tip: Editing a wall takes attached walls into account.

Tip: Deleting a wall deletes all doors and windows attached to it.



Edit windows and doors

Now let's play around with windows and doors. As we did for the walls, we can manipulate windows and doors more precisely in a few different ways.

In the 2D View, click on the feature to be edited.

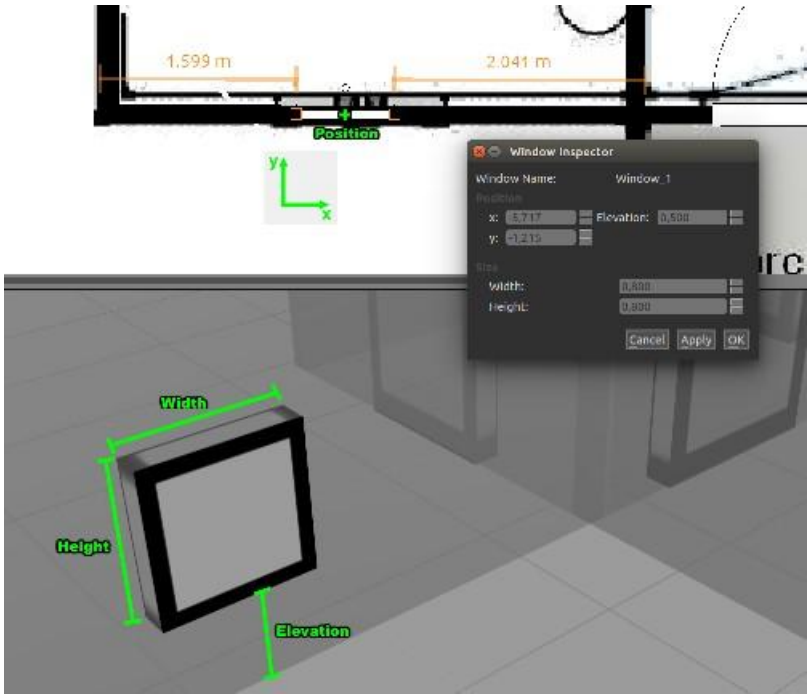
a. Translate the feature by dragging it to a new position. Remember that windows and doors automatically snap to walls and it doesn't make much sense to have them detached from any walls, as they represent holes in a wall.

b. Rotate the feature by dragging its rotation handle. Currently, as long as they are attached to a wall, their orientation doesn't make a difference.

c. Resize the feature's width by dragging one of the end points.

Double-click a feature in the 2D View to open an inspector with configuration options. Alternatively, right-click and choose Open Window/Door Inspector.

To delete a feature, either press the Delete key while it is selected, or right-click it in the 2D View and choose Delete.



Edit stairs

Finally, let's edit the staircase we inserted earlier. Since it is not on the floor plan, we can get creative and resize it as we want.

In the 2D View, click on the staircase to select it.

a. Translate the staircase by dragging it to a new position.

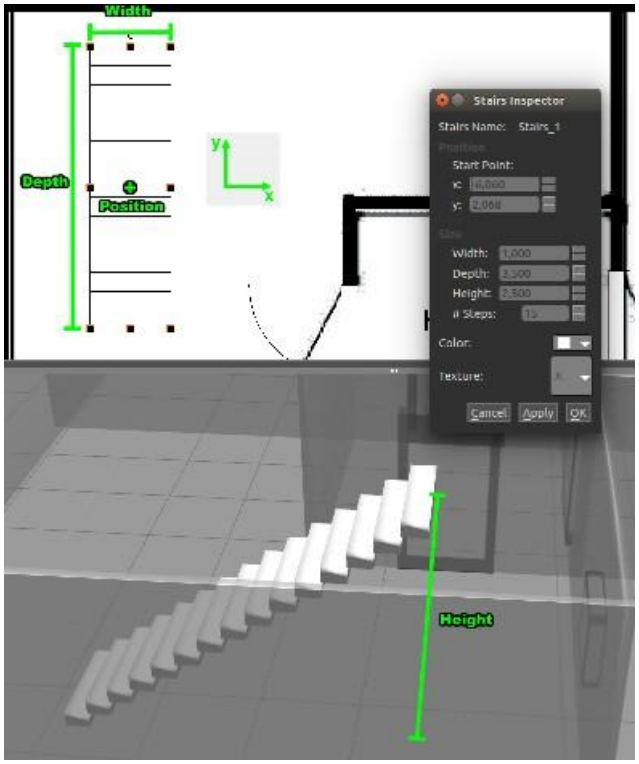
b. Rotate the staircase in multiples of 90° by dragging its rotation handle.

c. Resize the staircase by dragging one of the end nodes.

Double-click the staircase in the 2D View to open an inspector with configuration options. Alternatively, right-click and choose Open Stairs Inspector.

To delete the staircase, either press the Delete key while it is selected, or right-click and choose Delete.

Tip: In the 2D View, staircases are visible on both the start and end levels.



Add colors and textures

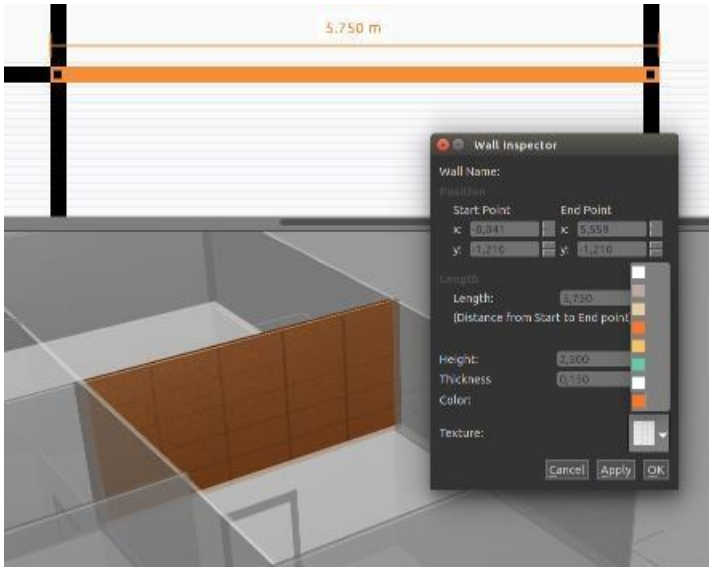
Now that everything is properly placed and sized, you can assign colors and textures to walls, floors and staircases. Remember that windows and doors are only holes on the wall and therefore cannot have materials.

Tip: The default color is white and the default texture is none.

There are two ways to add colors and textures to your building:

From Inspectors

You can add color and texture to walls, stairs and floors from the Wall Inspector, Stairs Inspector and Level Inspector respectively. Simply open the inspector, select your materials and press Apply.



From the Palette

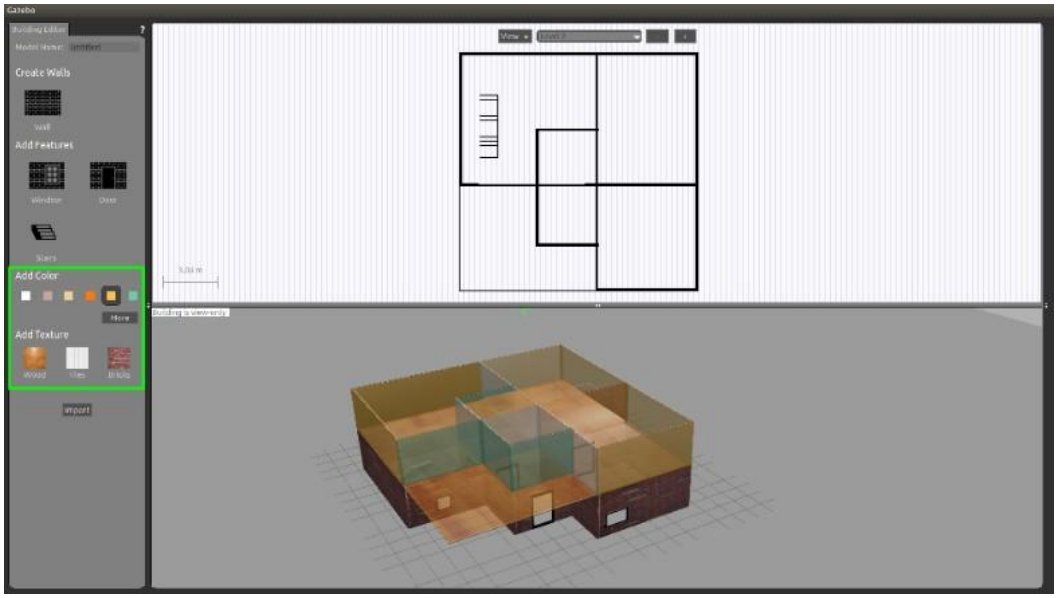
Colors and textures can be chosen from the Palette and assigned to items on your building by clicking on them in the 3D View.

Click on a color or texture in the Palette.

As you move your mouse in the 3D View, hovered features will be highlighted displaying a preview of the selected material.

Clicking on the highlighted feature assigns the selected material to it. You can click on as many features as you'd like.

When you're done with the selected material, either right-click the 3D view, or click outside any features to leave the material mode.



To choose a custom color, click on More on the Palette. A dialog opens where you can specify custom colors.

Tip: Each feature can have only one color and one texture. The same material is assigned to all faces of the feature.

Saving your building

Saving will create a directory, SDF and config files for your building.

Before saving, give your building a name on the Palette.

On the top menu, choose File, then Save As (or hit Ctrl+S). A dialog will come up where you can choose the location for your model.

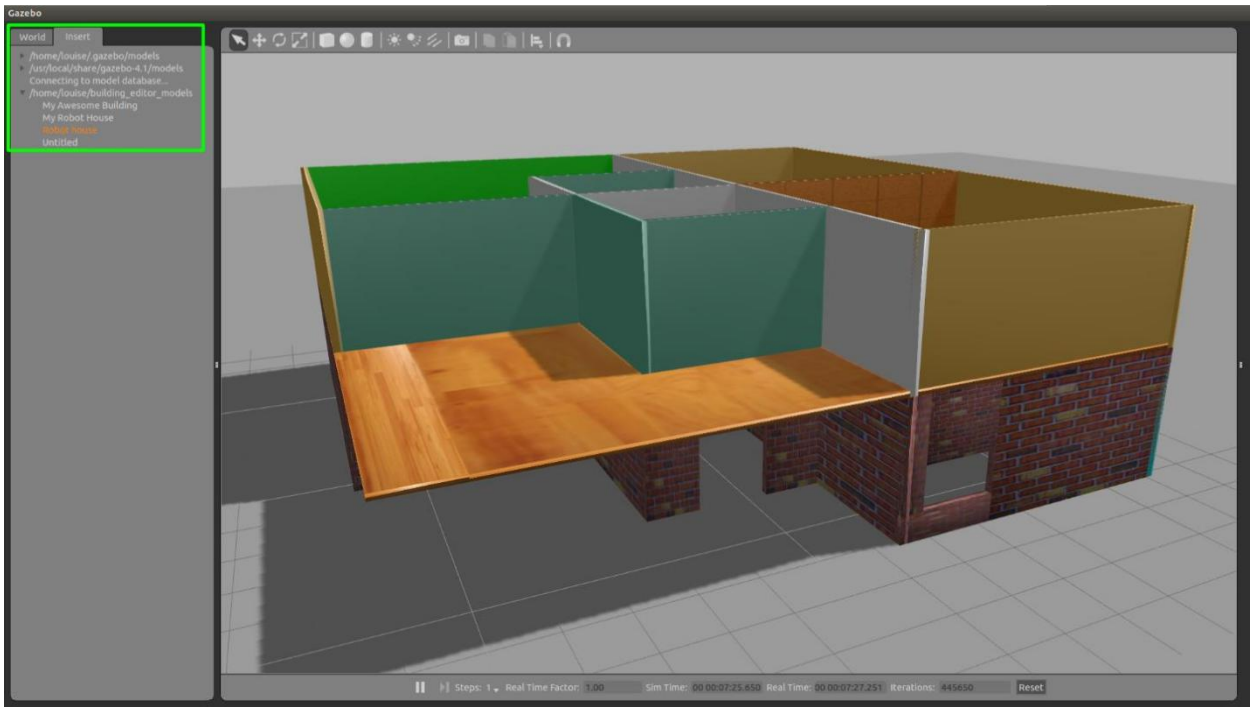
Tip: Under Advanced Options you can set some meta-data for your building.

Exit

Note: Once you exit the Building Editor, your building will no longer be editable.

When you're done creating your building and you've saved it, go to File and then Exit Building Editor.

Your building will show up in the main window. In the future, you can find the building in your Insert tab.



7.2 Add Sensor plugins like Laser, Kinect, etc. to URDF of mobile robot

Gazebo plugins give your URDF models greater functionality and can tie in ROS messages and service calls for sensor output and motor input.

Plugin Types

There are currently 6 types of plugins

1. World
2. Model
3. Sensor
4. System
5. Visual
6. GUI

Adding a SensorPlugin

Specifying sensor plugins is slightly different. Sensors in Gazebo are meant to be attached to links, so the <gazebo> element describing that sensor must be given a reference to that link. For example:

```
<robot>
  ... robot description ...
  <link name="sensor_link">
    ... link description ...
  </link>

  <gazebo reference="sensor_link">
    <sensor type="     " name="    ">
      ... sensor parameters ...
      <plugin name="        " filename="     ">
        ... plugin parameters ..
      </plugin>
    </sensor>
  </gazebo>
```

</robot

custom_bot – Three Wheel Robot with Laser Scanner

This Robot will scan the obstacles present in its environment

STEP:1

Path to be followed

catkin_ws---src--lab_bots--src--urdf

(make these directories and build the package by using catkin_make)

cd ~/catkin_ws/src/lab_bots/src/urdf

gedit custom_bot.urdf (copy the code on this text editor)

CODE:

```
<?xml version="1.0"?>
<robot name="gbj_bot">
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.6 0.3 0.1"/>
      </geometry>
      <material name="red">
        <color rgba="1 0.0 0.0 1"/>
      </material>
    </visual>
```

```
    <collision>
      <geometry>
        <box size="0.6 0.3 0.1"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="1.0"/>
      <inertia ixx="0.015" ixy="0" ixz="0" iyy="0.0375" iyz="0.0" izz="0.0375"/>
    </inertial>
  </link>

  <link name="front_caster_of_wheel">
    <visual>
      <geometry>
        <box size="0.1 0.1 0.1"/>
      </geometry>
      <material name="green">
        <color rgba="0.0 0.1 0.0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <box size="0.1 0.1 0.1"/>
      </geometry>
```

```
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="0.00083" ixy="0" ixz="0" iyy="0.00083" iyz="0.0" izz="0.000167"/>
    </inertial>
  </link>
  <joint name="front_caster_of_wheel_joint" type="continuous">
    <axis xyz="0.0 0.0 1"/>
    <parent link="base_link"/>
    <child link="front_caster_of_wheel"/>
    <origin xyz="0.3 0.0 0.0" rpy="0.0 0.0 0.0"/>
  </joint>
  <link name="front_wheel">
     <visual>
       <geometry>
         <cylinder radius="0.035" length="0.05"/>
       </geometry>
       <material name="black">
       </material>
     </visual>
     <collision>
       <geometry>
         <cylinder radius="0.035" length="0.05"/>
       </geometry>
```

```
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="5.1458e-5" ixy="0" ixz="0" iyy="5.1458e-5" iyz="0.0" izz="6.125e-5"/>
    </inertial>
  </link>
  <joint name="front_wheel_joint" type="continuous">
    <axis xyz="0.0 0.0 1"/>
    <parent link="front_caster_of_wheel"/>
    <child link="front_wheel"/>
    <origin xyz="0.05 0.0 -0.05" rpy="-1.5708 0.0 0.0"/>
  </joint>

  <link name="right_wheel">
    <visual>
      <geometry>
        <cylinder radius="0.035" length="0.05"/>
      </geometry>
      <material name="black">
        <color rgba="0.0 0.0 0.0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
```

```
            <cylinder radius="0.035" length="0.05"/>
        </geometry>
    </collision>
    <inertial>
        <mass value="0.1"/>
        <inertia ixx="5.1458e-5" ixy="0" ixz="0" iyy="5.1458e-5" iyz="0.0" izz="6.125e-5"/>
    </inertial>
</link>
<joint name="right_wheel_joint" type="continuous">
    <axis xyz="0.0 0.0 1"/>
    <parent link="base_link"/>
    <child link="right_wheel"/>
    <origin xyz="-0.2825 -0.125 -0.05" rpy="-1.5708 0.0 0.0"/>
</joint>


<link name="left_wheel">
    <visual>
        <geometry>
            <cylinder radius="0.035" length="0.05"/>
        </geometry>
        <material name="black">
            <color rgba="0.0 0.0 0.0 1"/>
        </material>
    </visual>
```

```
    <collision>
      <geometry>
        <cylinder radius="0.035" length="0.05"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="5.1458e-5" ixy="0" ixz="0" iyy="5.1458e-5" iyz="0.0" izz="6.125e-5"/>
   </inertial>
</link>
<joint name="left_wheel_joint" type="continuous">
    <axis xyz="0.0 0.0 1"/>
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin xyz="-0.2825 0.125 -0.05" rpy="-1.5708 0.0 0.0"/>
</joint>


<gazebo>
    <plugin name="differential_drive_controller"
          filename="libgazebo_ros_diff_drive.so">
      <leftJoint>left_wheel_joint</leftJoint>
      <rightJoint>right_wheel_joint</rightJoint>
      <legacyMode>false</legacyMode>
      <robotBaseFrame>base_link</robotBaseFrame>
```

```
        <wheelSeparation>0.25</wheelSeparation>
        <wheelDiameter>0.07</wheelDiameter>
        <publishWheelJointState>true</publishWheelJointState>
    </plugin>
</gazebo>

<gazebo>
    <plugin name="joint_state_publisher"
            filename="libgazebo_ros_joint_state_publisher.so">
        <jointName>front_caster_of_wheel_joint, front_wheel_joint</jointName>
    </plugin>
</gazebo>

<link name="laser_scanner">
    <visual>
        <geometry>
            <box size="0.1 0.1 0.1"/>
        </geometry>
    </visual>

    <collision>
        <geometry>
            <box size="0.1 0.1 0.1"/>
        </geometry>
```

```
    </collision>

    <inertial>
      <mass value="1e-5"/>
      <inertia ixx="1e-6" ixy="0" ixz="0.0" iyy="1e-6" iyz="0.0" izz="1e-6"/>
    </inertial>
  </link>

  <joint name="laser_scanner_joint" type="fixed">
    <axis xyz="0.0 1 0.0"/>
    <parent link="base_link"/>
    <child link="laser_scanner"/>
    <origin xyz="0.0 0.0 0.08" rpy="0.0 0.0 0.0"/>
  </joint>

  <gazebo reference="laser_scanner">
    <sensor type="gpu_ray" name="laser">
      <pose>0 0 0 0 0 0</pose>
      <visualize>false</visualize>
      <update_rate>40</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>720</samples>
```

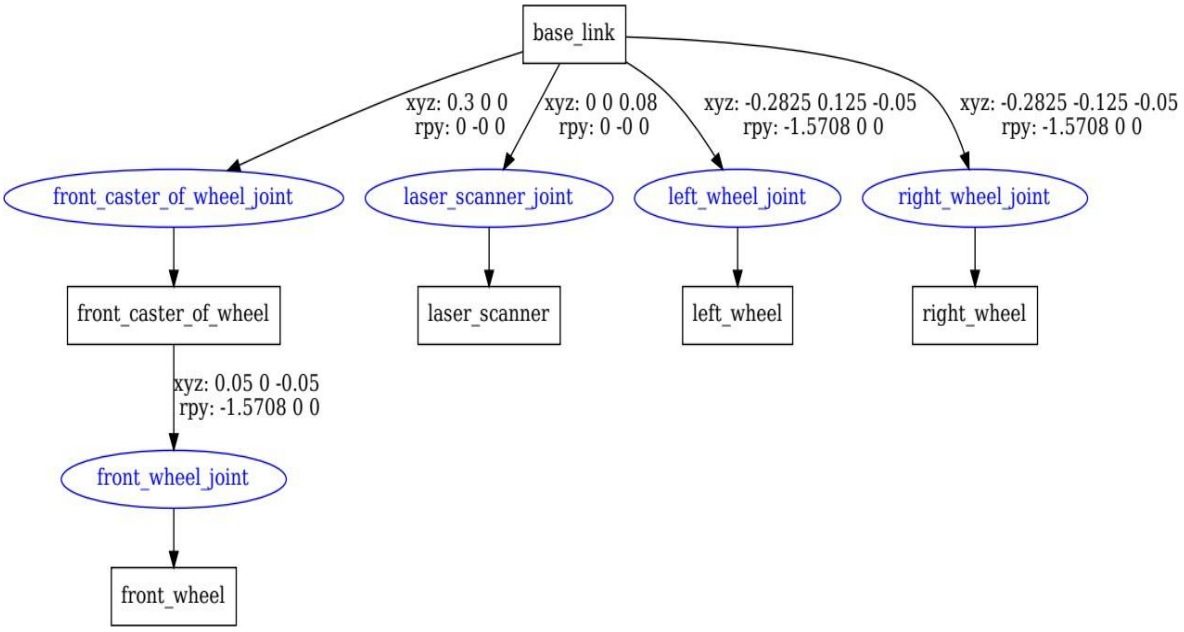
```
            <resolution>1</resolution>
            <min_angle>-1.578</min_angle>
            <max_angle>1.578</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.1</min>
          <max>30</max>
          <resolution>0.1</resolution>
        </range>
      </ray>
      <plugin name="gpu_laser" filename="libgazebo_ros_gpu_laser.so">
        <topicName>/scan</topicName>
        <frameName>laser_scanner</frameName>
      </plugin>
    </sensor>
  </gazebo>
  </robot>
```

STEP:2

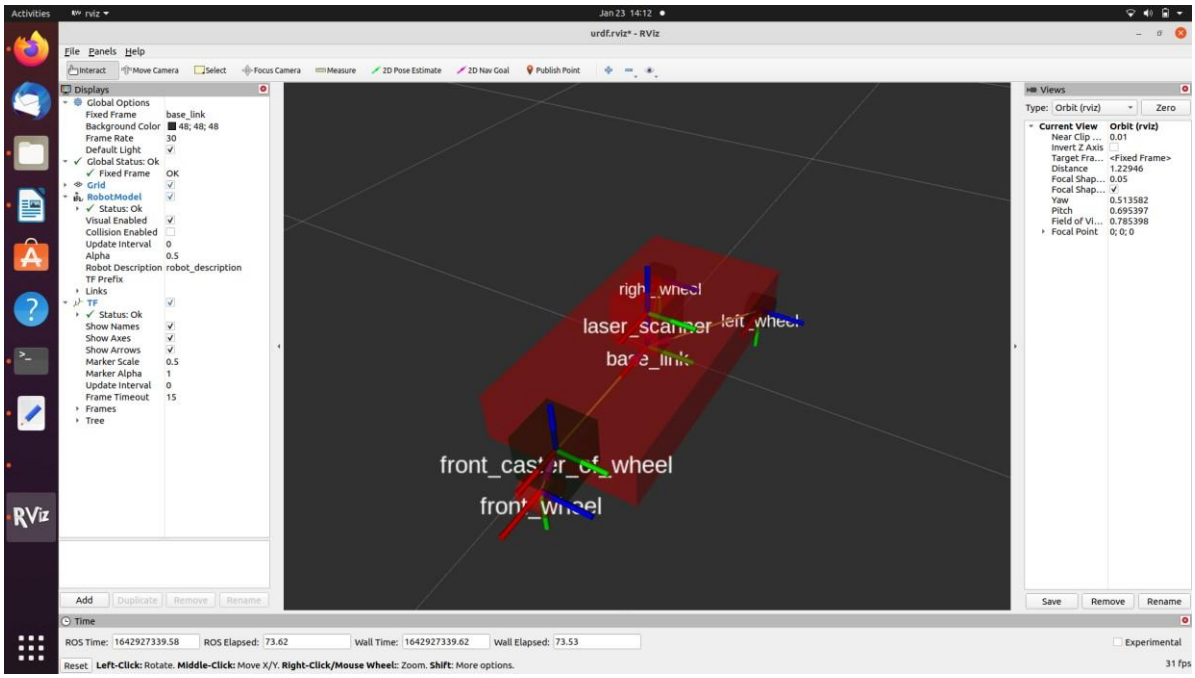
To see the build model in URDF follow the given commands:

cd ~/catkin_ws/src/lab_bots/src/urdf

urdf_to_graphiz custom_bot.urdf (for Visualizing the structure of the Robot)


base_link
xyz: 0.3 0 0
rpy: 0 -0 0
xyz: 0 0 0.08
rpy: 0 -0 0
xyz: -0.2825 0.125 -0.05
rpy: -1.5708 0 0
xyz: -0.2825 -0.125 -0.05
rpy: -1.5708 0 0
front_caster_of_wheel_joint
laser_scanner_joint
left_wheel_joint
right_wheel_joint
front_caster_of_wheel
laser_scanner
left_wheel
right_wheel
xyz: 0.05 0 -0.05
rpy: -1.5708 0 0
front_wheel_joint
front_wheel

roslaunch urdf_tutorial display.launch model:='custom_bot.urdf'



STEP:3 -RUN the code

Copy config folder to lab_bots package (to get any folder in Windows – right click on folder and click on Reveal in Explorer)

Packages need to install (make sure that these packages are present in your distribution)

slam_gmapping -The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping. Using slam_gmapping, you can create a 2-D occupancy grid map **(like a building floorplan) from laser and pose data collected by a mobile robot**.

teleop_twist_keyboard- This node is constantly checking which keys are pressed on a PC keyboard and based on the keys pressed, publishes twist messages on /cmd_vel topic . Twist message defines what should be the linear and rotational speeds of a mobile robot.

robot_state_publisher- publishes the **tf transforms of your robot based on** its URDF file. You can also publish the joint states using the joint_state_publisher or do it yourself. Either way, the robot_state_publisher uses this information to calculate the forward kinematics of your robot

ROS Service Call" Robot Spawn Method-This method uses a small python script called spawn_model to make a service call request to the gazebo_ros ROS node (named simply

"gazebo" in the rostopic namespace) to add a custom URDF into Gazebo. The spawn_model script is located within the gazebo_ros package.

move_base-*This* node provides a *ROS* interface for configuring, running, and interacting with the navigation stack on a robot

To Run the code make this path

cd ~/catkin_ws/src/lab_bots/src/launch

gedit bot_model.launch

CODE:

```
<launch>

  <param name="robot_description"
textfile="/home/roshni/catkin_ws/src/lab_bots/src/urdf/custom_bot.urdf" />

 <include file="$(find gazebo_ros)/launch/empty_world.launch">


  </include>

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" />


  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"

      args="-param robot_description -urdf -model custom_bot" />


  <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
name="teleop_twist_keyboard" output="screen"/>

  <node pkg="rviz" type="rviz" name="rviz"/>
```

```
  <node pkg="gmapping" type="slam_gmapping" name="gmapping">
    <param name="base_frame" value="base_link" />
    <param name="odom_frame" value="odom" />
    <param name="delta" value="0.1" />
  </node>

  <node pkg="move_base" type="move_base" name="move_base" output="screen">
    <param name="controller_frequency" value="10.0" />
    <rosparam file="/home/roshni/catkin_ws/src/lab_bots/config/costmap_common_params.yaml" command="load" ns="global_costmap"/>
    <rosparam file="/home/roshni/catkin_ws/src/lab_bots/config/costmap_common_params.yaml" command="load" ns="local_costmap"/>
    <rosparam file="/home/roshni/catkin_ws/src/lab_bots/config/local_costmap_params.yaml" command="load" />
    <rosparam file="/home/roshni/catkin_ws/src/lab_bots/config/global_costmap_params.yaml" command="load" />
    <rosparam file="/home/roshni/catkin_ws/src/lab_bots/config/trajectory_planner.yaml" command="load" />

  </node>
</launch>
```

The <rosparam> tag enables the use of HYPERLINK "http://wiki.ros.org/rosparam" \h rosparam YAML files for loading and dumping parameters from the ROS Parameter Server. It can also be used to remove parameters. The <rosparam> tag can be put inside of a <node> tag, in which case the parameter is treated like a private name

cd ~/catkin_ws/src/lab_bots/src/launch

roslaunch bot_model.launch

RVIZ may show error: Change fixed frame to odom even if it doesn't show error

If it is still not working launch RVIZ separately as below

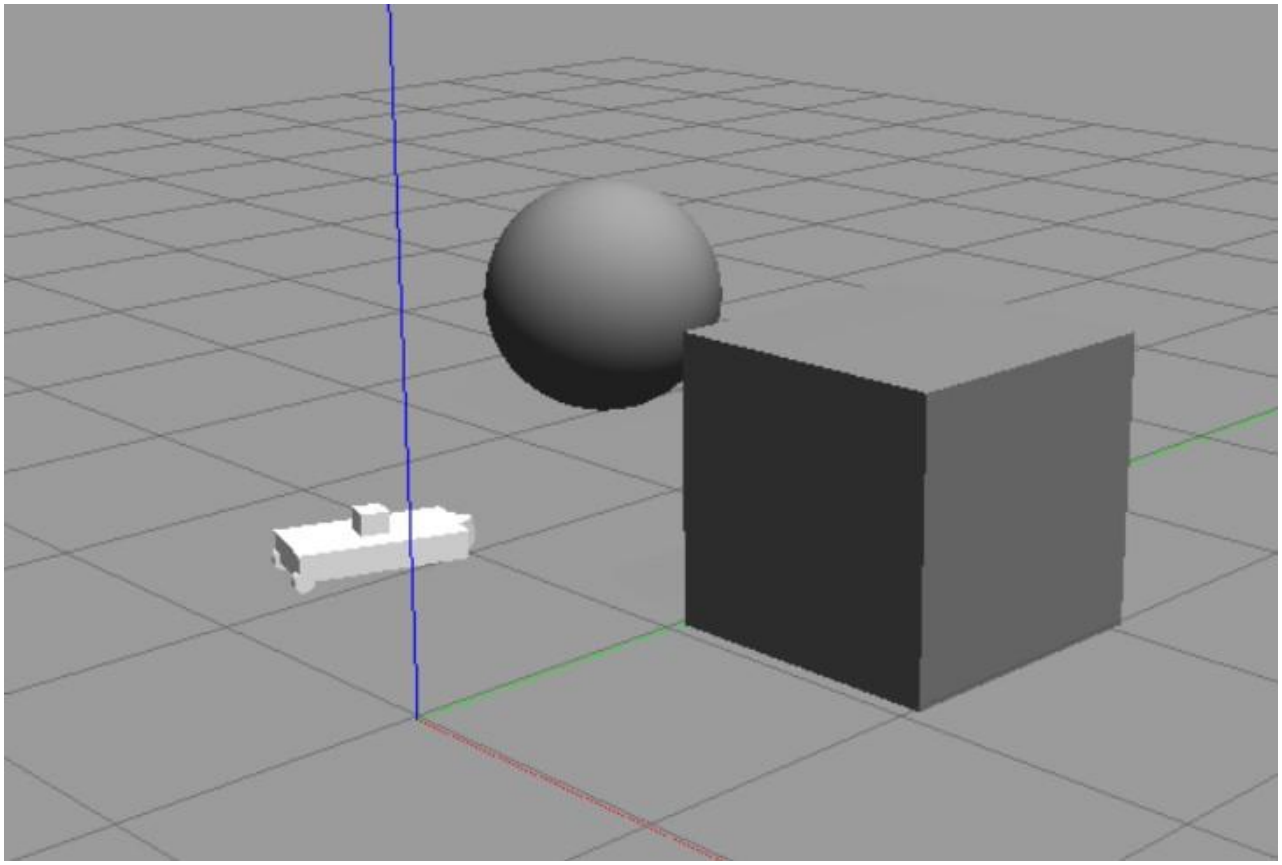
cd ~/catkin_ws/src/lab-bots/src/urdf

roslaunch urdf_tutorial display.launch model:='custom_bot.urdf'

In RVIZ add Robot model and Laser Scanner. In Laser Scanner change topic to /scan

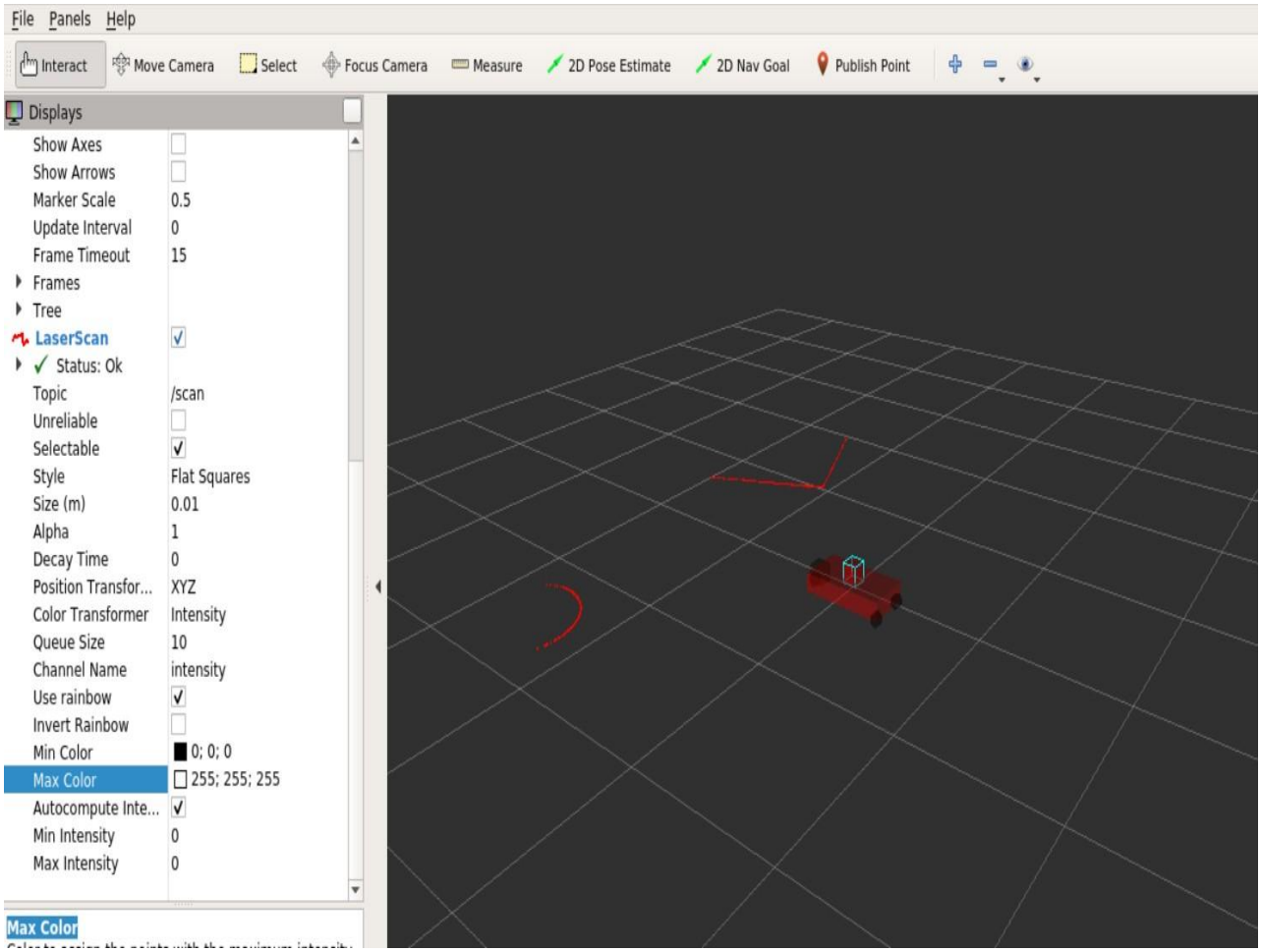
Adding blocks in front of robot in Gazebo, we can visualize its scanned image in Rviz

GAZEBO



Move robot using keys specifiedCan use Gazebo – Edit – Reset Model Poses to bring robot back to original pose

RVIZ


File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Show Axes
Show Arrows
Marker Scale 0.5
Update Interval 0
Frame Timeout 15
Frames
Tree
LaserScan
Status: Ok
Topic /scan
Unreliable
Selectable
Style Flat Squares
Size (m) 0.01
Alpha 1
Decay Time 0
Position Transfor... XYZ
Color Transformer Intensity
Queue Size 10
Channel Name intensity
Use rainbow
Invert Rainbow
Min Color 0; 0; 0
Max Color 255; 255; 255
Autocompute Inte...
Min Intensity 0
Max Intensity 0
Max Color

RESULT:

Experiment.8

Create a 3DOF robotic arm from scratch

Instead of 3 DOF Robot arm, the code below is for a 7 DOF Robot arm – this is so that the same code can be used for Experiment 10 - MOVEIT

CODE:

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

<robot name="robot_arm">



  <material name="Black">
    <color rgba="0.0 0.0 0.0 1.0"/>
  </material>
  <material name="Red">
    <color rgba="0.8 0.0 0.0 1.0"/>
  </material>
  <material name="White">
    <color rgba="1.0 1.0 1.0 1.0"/>
  </material>
  <material name="Green">
    <color rgba="0.0 1.0 0.0 1.0"/>
  </material>
  <material name="Blue">
    <color rgba="0.0 0.0 0.8 1.0"/>
  </material>
```

```
<material name="Yellow">
   <color rgba="1.0 1.0 0.0 1.0"/>
</material>

<link name="base_link">
   <visual>
      <origin rpy="1.570795 0 0" xyz="0 0 0"/>
      <!-- rotate PI/2 -->
      <geometry>
         <box size="0.1 0.1 0.1"/>
      </geometry>
      <material name="White"/>
   </visual>
</link>

<link name="bottom_link">
   <visual>
      <origin rpy="0 0 0" xyz=" 0 0 0"/>
      <geometry>
         <box size="0.5 0.5 0.02"/>
      </geometry>
      <material name="Red"/>
   </visual>
</link>
```

```
<joint name="bottom_joint" type="fixed">
   <origin rpy="0 0 0" xyz="0 0 -0.04"/>
   <parent link="base_link"/>
   <child link="bottom_link"/>
</joint>

<link name="shoulder_pan_link">
   <visual>
      <origin rpy="0 1.570795 0" xyz="0 0 0"/>
      <geometry>
         <cylinder length="0.08" radius="0.04"/>
      </geometry>
      <material name="Green"/>
   </visual>
</link>

<joint name="shoulder_pan_joint" type="revolute">
   <parent link="base_link"/>
   <child link="shoulder_pan_link"/>
   <origin rpy="0 1.570795 0.0" xyz="0 0 0.05"/>
   <axis xyz="-1 0 0"/>
   <limit effort="300" lower="-2.61799387799" upper="1.98394848567" velocity="1"/>
</joint>
```

```
<link name="shoulder_pitch_link">
   <visual>
      <origin rpy="0 1.570795 0" xyz="0 0 0.04"/>
      <geometry>
         <box size="0.14 0.04 0.04"/>
      </geometry>
      <material name="Red"/>
   </visual>
</link>

<joint name="shoulder_pitch_joint" type="revolute">
   <parent link="shoulder_pan_link"/>
   <child link="shoulder_pitch_link"/>
   <origin rpy="-1.570795 0 1.570795" xyz="-0.04 0.002 0.0"/>
   <axis xyz="1 0 0"/>
   <limit effort="300" lower="-1.19962513147" upper="1.89994105047" velocity="1"/>
</joint>

<link name="elbow_roll_link">
   <visual>
      <origin rpy="0 1.570795 0" xyz="0.0 0.0 0"/>
      <geometry>
         <cylinder length="0.04" radius="0.03"/>
```

```
        </geometry>
        <material name="Black"/>
    </visual>
</link>

<joint name="elbow_roll_joint" type="revolute">
    <parent link="shoulder_pitch_link"/>
    <child link="elbow_roll_link"/>
    <origin rpy="0 1.570795 0.0" xyz="0.0 0 0.12"/>
    <axis xyz="-1 0 0"/>
    <limit effort="300" lower="-2.61799387799" upper="1.98394848567" velocity="1"/>
</joint>

<link name="elbow_pitch_link">
    <visual>
        <origin rpy="0 1.570795 0" xyz="0 0 -0.08"/>
        <geometry>
            <box size="0.14 0.04 0.04"/>
        </geometry>
        <material name="Blue"/>
    </visual>
</link>

<joint name="elbow_pitch_joint" type="revolute">
```

```
    <parent link="elbow_roll_link"/>
    <child link="elbow_pitch_link"/>
    <origin rpy="0 1.570795 0" xyz="0 0 0"/>
    <axis xyz="1 0 0"/>
    <limit effort="300" lower="-1.5953400194" upper="1.93281579274" velocity="1"/>
</joint>

<link name="wrist_roll_link">
    <visual>
        <origin rpy="0 1.570795 0" xyz="0 0 0"/>
        <geometry>
            <cylinder length="0.02" radius="0.02"/>
        </geometry>
        <material name="Green"/>
    </visual>
</link>

<joint name="wrist_roll_joint" type="revolute">
    <parent link="elbow_pitch_link"/>
    <child link="wrist_roll_link"/>
    <origin rpy="0 1.570795 3.14159" xyz="0 0 -0.16"/>
    <axis xyz="1 0 0"/>
    <limit effort="300" lower="-2.61799387799" upper="2.6128806087" velocity="1"/>
</joint>
```

```
<link name="gripper_finger_link1">
    <visual>
      <origin xyz="0 0 0"/>
      <geometry>
        <box size="0.08 0.01 0.01"/>
      </geometry>
      <material name="Yellow"/>
    </visual>
  </link>

  <joint name="finger_joint1" type="prismatic">
    <parent link="wrist_roll_link"/>
    <child link="gripper_finger_link1"/>
    <origin xyz="0.04 -0.02 0"/>
    <axis xyz="0 1 0"/>
    <limit effort="100" lower="0" upper="0.03" velocity="1.0"/>
  </joint>

  <link name="gripper_finger_link2">
    <visual>
      <origin xyz="0.0 0.0 0"/>
      <geometry>
        <box size="0.08 0.01 0.01"/>
      </geometry>
```

```
        <material name="Yellow"/>
      </visual>
    </link>

    <joint name="finger_joint2" type="prismatic">
      <parent link="wrist_roll_link"/>
      <child link="gripper_finger_link2"/>
      <origin xyz="0.04 0.02 0"/>
      <axis xyz="0 1 0"/>
      <limit effort="1" lower="-0.03" upper="0" velocity="1.0"/>
    </joint>
</robot>
```

cd ~/catkin_ws/src/lab_bots/src/urdf

roslaunch urdf_tutorial display.launch model:='robot_arm.urdf'

RESULT:

Experiment.9

AIM: Familiarisation with MoveIt through its RViz plugin, Motion Planning with the Panda or other robot models.

MOVEIT

MoveIt is a primary source of the functionality for manipulation (and mobile manipulation) in ROS.

https://ros-planning.github.io/moveit_tutorials/doc/getting_started/getting_started.html

Installation of MOVEIT

sudo apt-get install ros-noetic-moveit

```
sudo apt install ros-noetic-desktop-full
```

```
sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator
python3-wstool build-essential
sudo apt install python3-rosdep
sudo rosdep init
rosdep update
```

Create A Catkin Workspace and Download MoveIt Source

mkdir -p ~/ws_moveit/src

cd ~/ws_moveit/src

wstool init .

wstool merge -t . https://raw.githubusercontent.com/ros-planning/moveit/master/moveit.rosinstall

wstool remove moveit_tutorials # this is cloned in the next section

wstool update -t .

Within your catkin workspace, download the moveit tutorials as well as the panda_moveit_config package

cd ~/ws_moveit/src

git clone https://github.com/ros-planning/moveit_tutorials.git -b master

git clone https://github.com/ros-planning/panda_moveit_config.git -b melodic-devel

The following will install from Debian any package dependencies not already in your workspace

cd ~/ws_moveit/src

rosdep install -y --from-paths . --ignore-src --rosdistro noetic

The next command will configure your catkin workspace:

cd ~/ws_moveit

catkin config --extend /opt/ros/${ROS_DISTRO} --cmake-args -DCMAKE_BUILD_TYPE=Release

catkin_make

Source the catkin workspace:

source ~/ws_moveit/devel/setup.bash

To launch MOVEIT setup_assistant

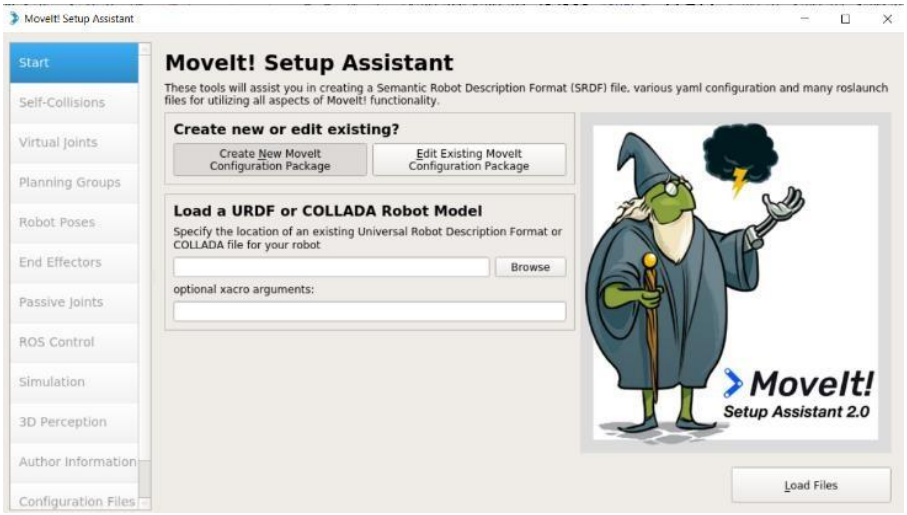
Step 1: Start

To start the MoveIt! Setup Assistant:

roslaunch moveit_setup_assistant setup_assistant.launch

This will bring up the start screen with two choices: Create New MoveIt! Configuration Package or Edit Existing MoveIt! Configuration Package.

Click on the Create New MoveIt! Configuration Package button to bring up the following screen:



Load Files – Load a urdf or xacro model using Browse – then click Load Files (Load robot_arm.urdf)

NB: load the URDF file created in the Experiment 8

Step 2: Generate Self-Collision Matrix

The Default Self-Collision Matrix Generator searches for pairs of links on the robot that can safely be disabled from collision checking, decreasing motion planning processing time.

The Default Self-Collision Matrix Generator searches for pairs of links on the robot that can safely be disabled from collision checking, decreasing motion planning processing time. These pairs of links are disabled when they are always in collision, never in collision, in collision in the robot's default position or when the links are adjacent to each other on the kinematic chain. The sampling density specifies how many random robot positions to check for self collision. Higher densities require more computation time while lower densities have a higher possibility of disabling pairs that should not be disabled. The default value is 10,000 collision checks. Collision checking is done in parallel to decrease processing time.

Click on the Self-Collisions pane selector on the left-hand side and click on the Generate Collision Matrix button. The Setup Assistant will work for a few second before presenting you the results of its computation in the main table.

Only adjacent or never in collision should be there.

Step 3: Add Virtual Joints

Virtual joints are used primarily to attach the robot to the world. For the robot, we will define only one virtual joint attaching the base_link of the robot to the world frame. This virtual joint represents the motion of the base of the robot in a plane.

Click on the Virtual Joints pane selector. Click on Add Virtual Joint

Set the joint name as “virtual_joint”

Set the child link as “base_link” and the parent frame name as “world”.

Set the Joint Type as “fixed”.

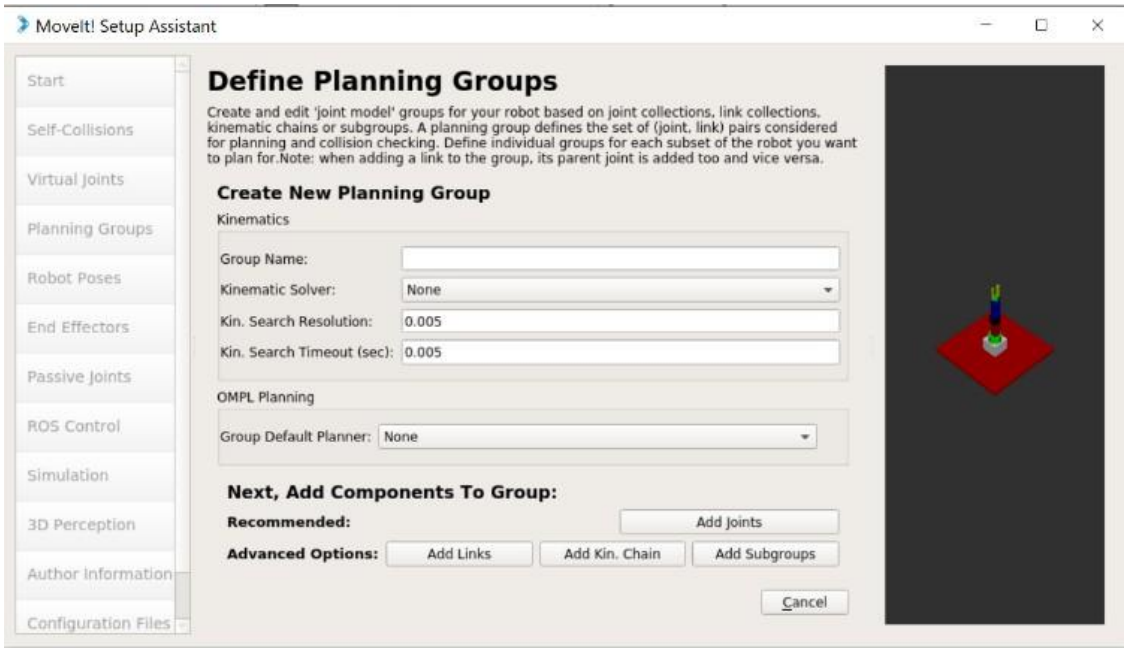
Click Save and you should see this screen:

Step 4: Add Planning Groups

Planning groups are used for semantically describing different parts of your robot, such as defining what an arm is, or an end effector.

Click on the Planning Groups pane selector.

Click on Add Group and you should see the following screen:



Add the arm

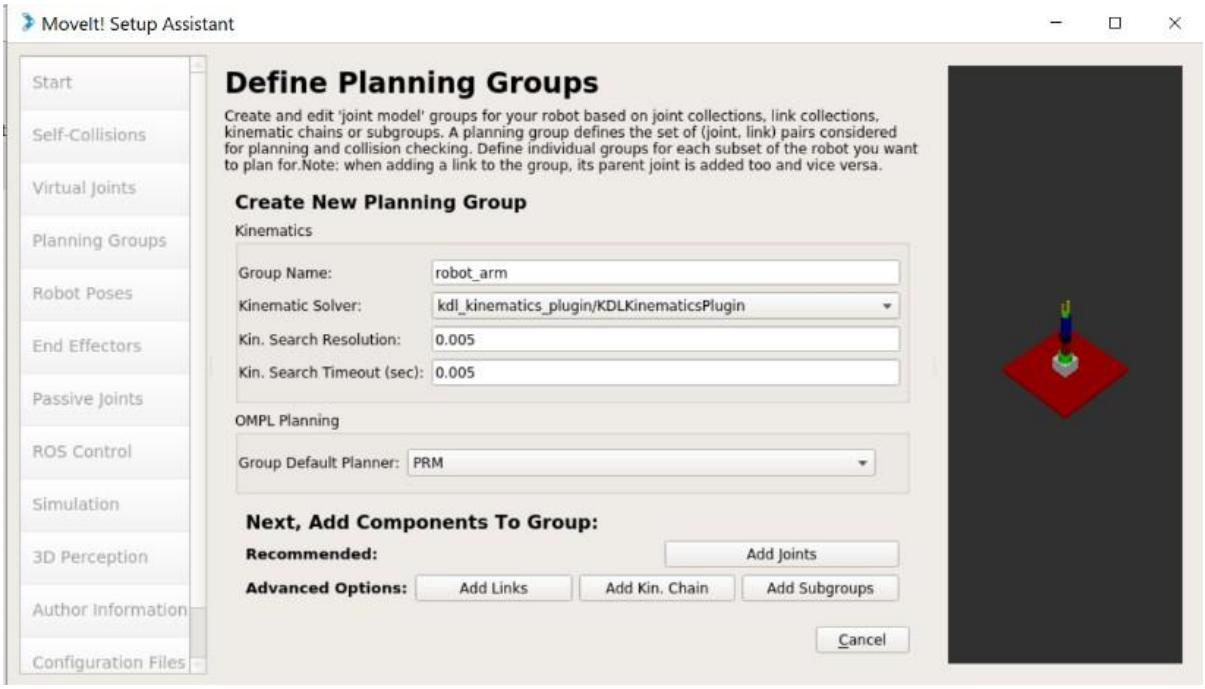
We will first add robot arm as a planning group

Enter Group Name as robot_arm

Choose kdl_kinematics_plugin/KDLKinematicsPlugin as the kinematics solver. Note: if you have a custom robot and would like a powerful custom IK solver, see Kinematics/IKFast

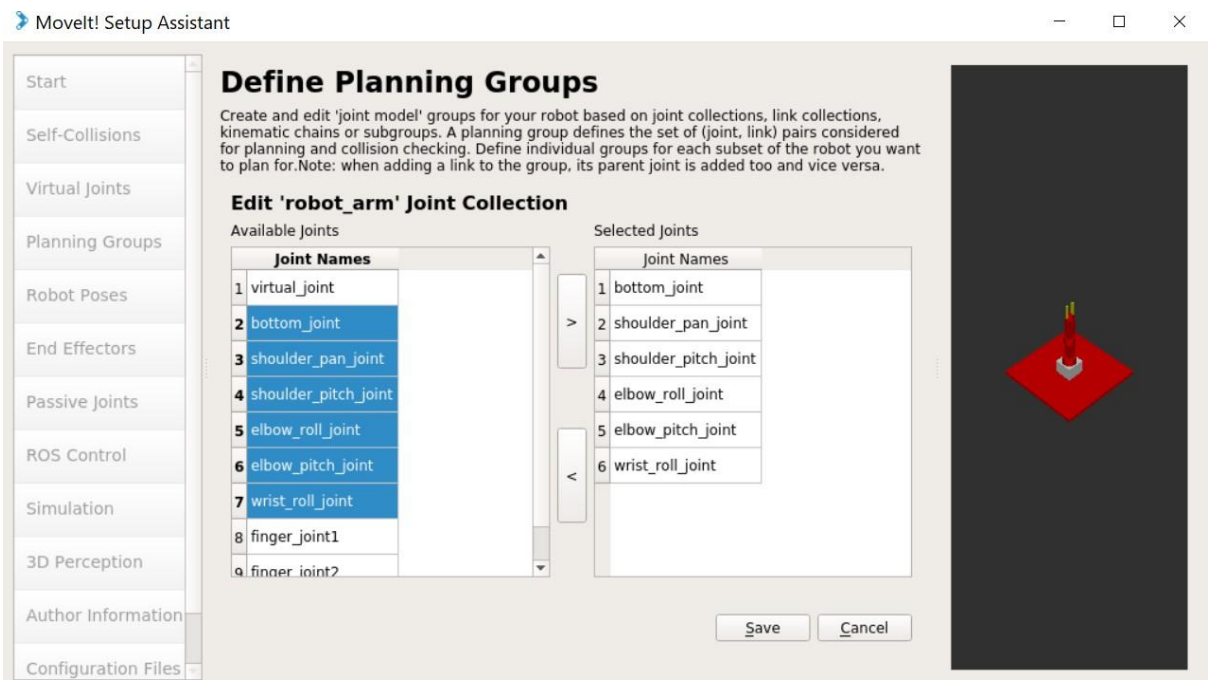
Let Kin. Search Resolution and Kin. Search Timeout stay at their default values.

Group Default Planner – PRM

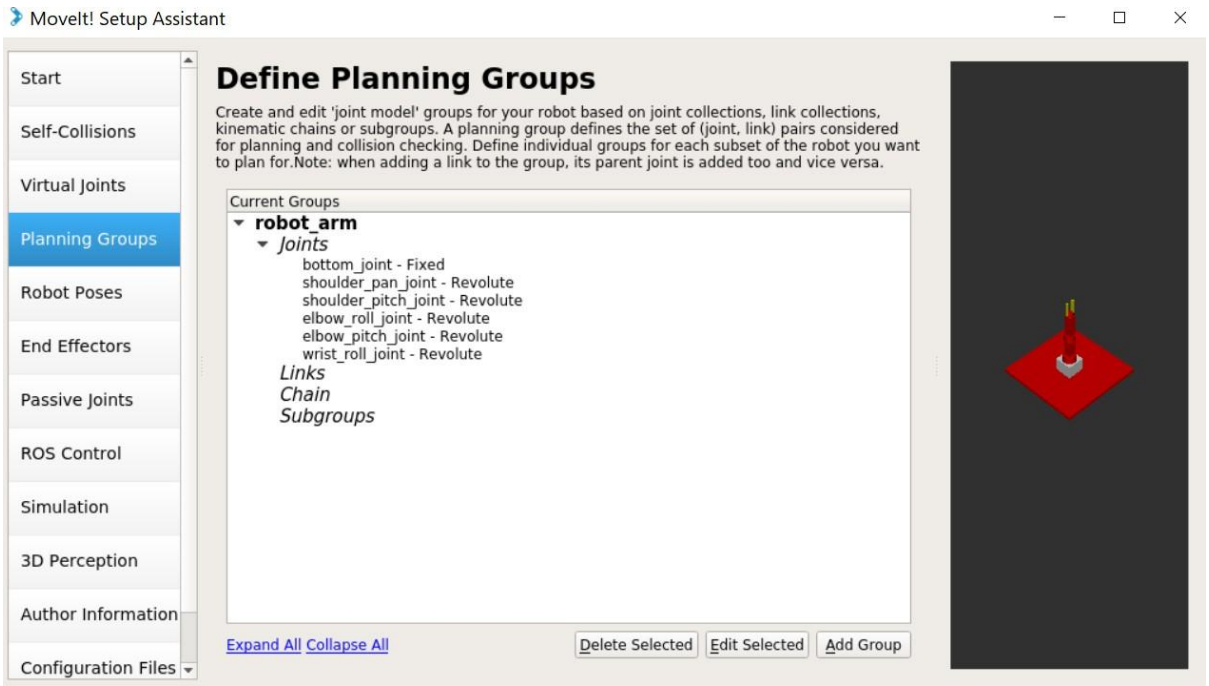


Now, click on the Add Joints button. You will see a list of joints on the left hand side. You need to choose all the joints that belong to the arm and add them to the right hand side. The joints are arranged in the order that they are stored in an internal tree structure. This makes it easy to select a serial chain of joints.

Click on the arm joints. Now click on the > button to add these joints into the list of selected joints on the right.



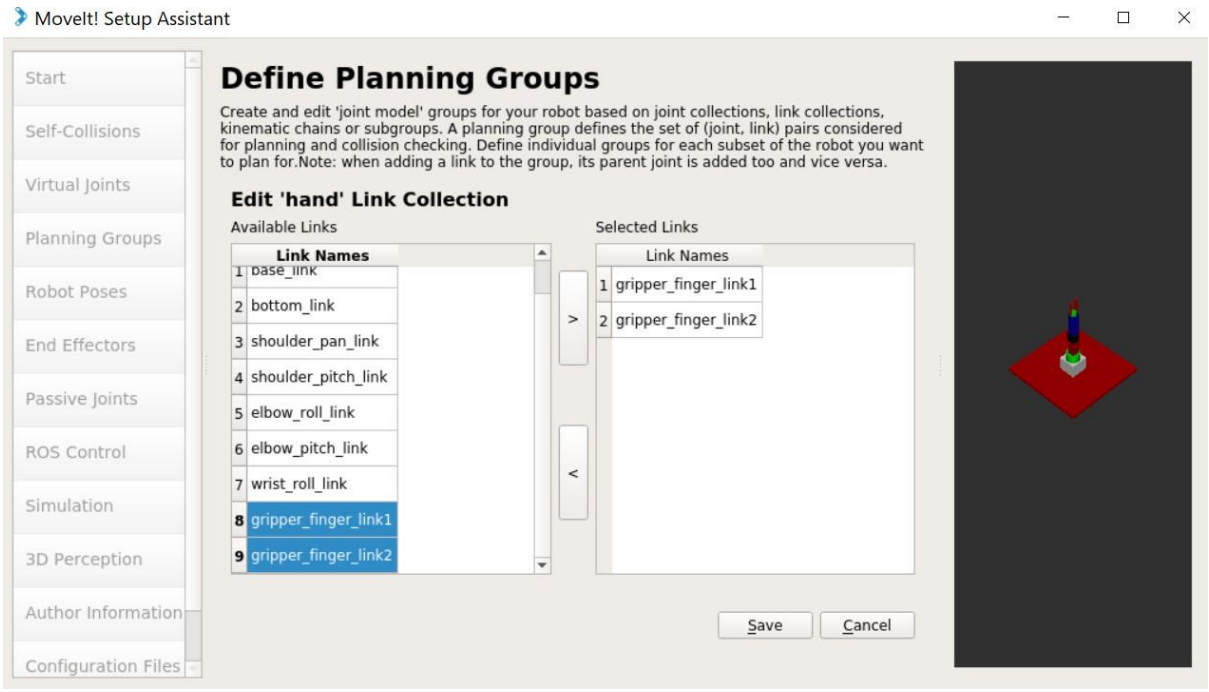
Click Save to save the selected group



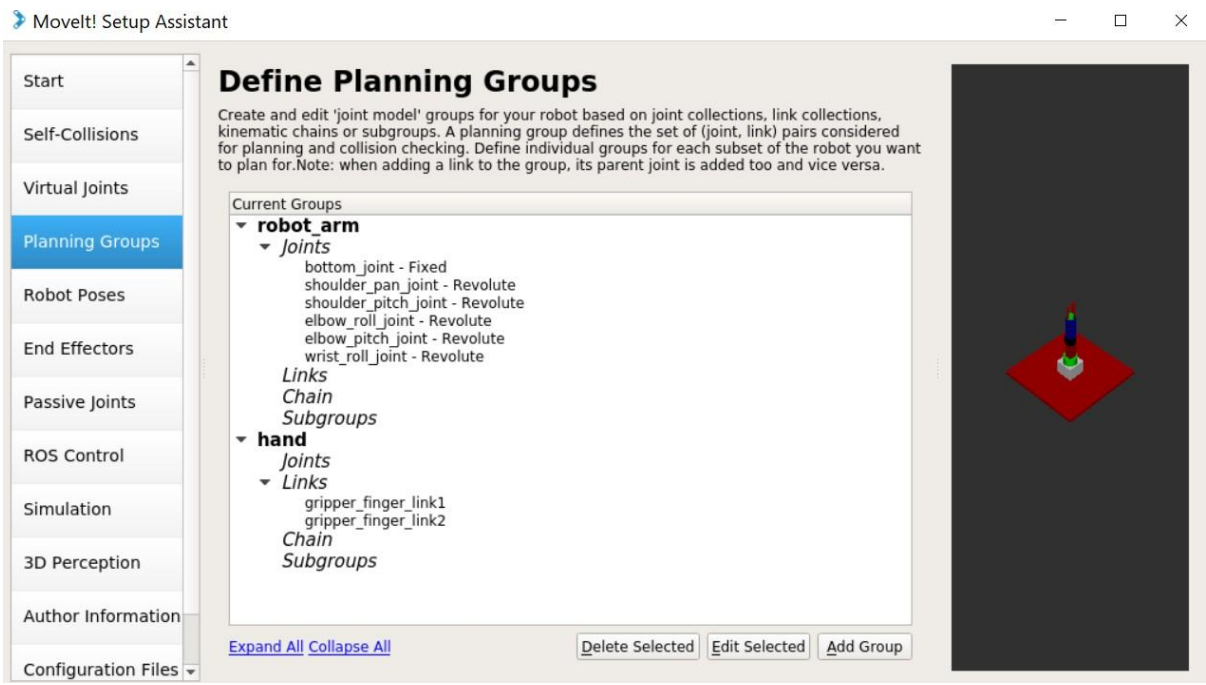
Add the gripper

- We will also add a group for the end effector. NOTE that you will do this using a different procedure than adding the arm.

- Click on the Add Group button.
- Enter Group Name as hand
- Let Kin. Search Resolution and Kin. Search Timeout stay at their default values.
- Click on the Add Links button.
- Choose gripper links and add them to the list of Selected Links on the right hand side.



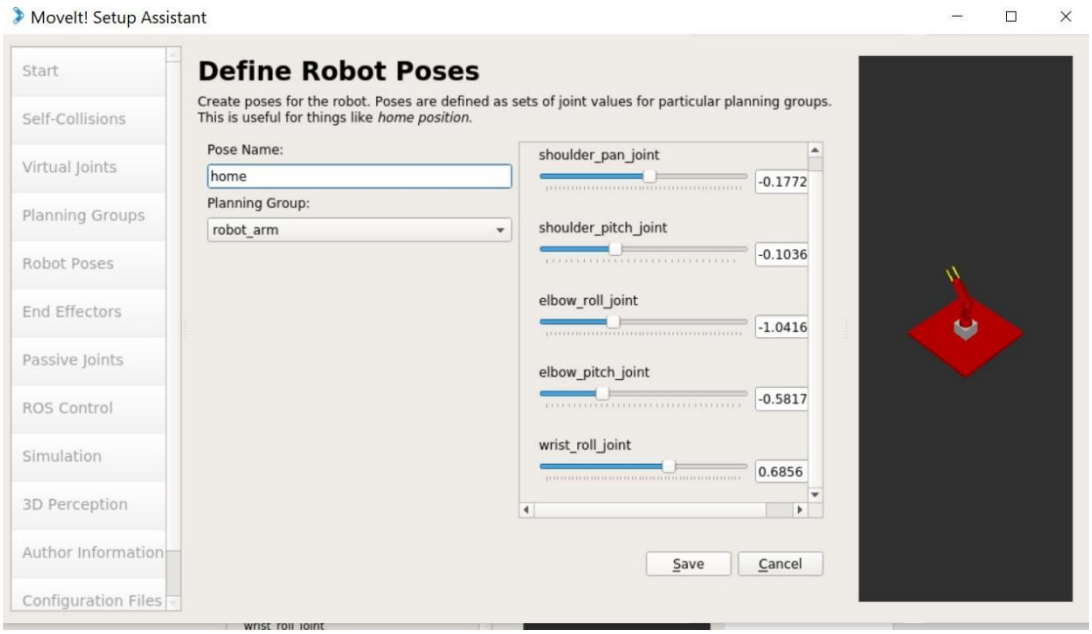
- Click Save

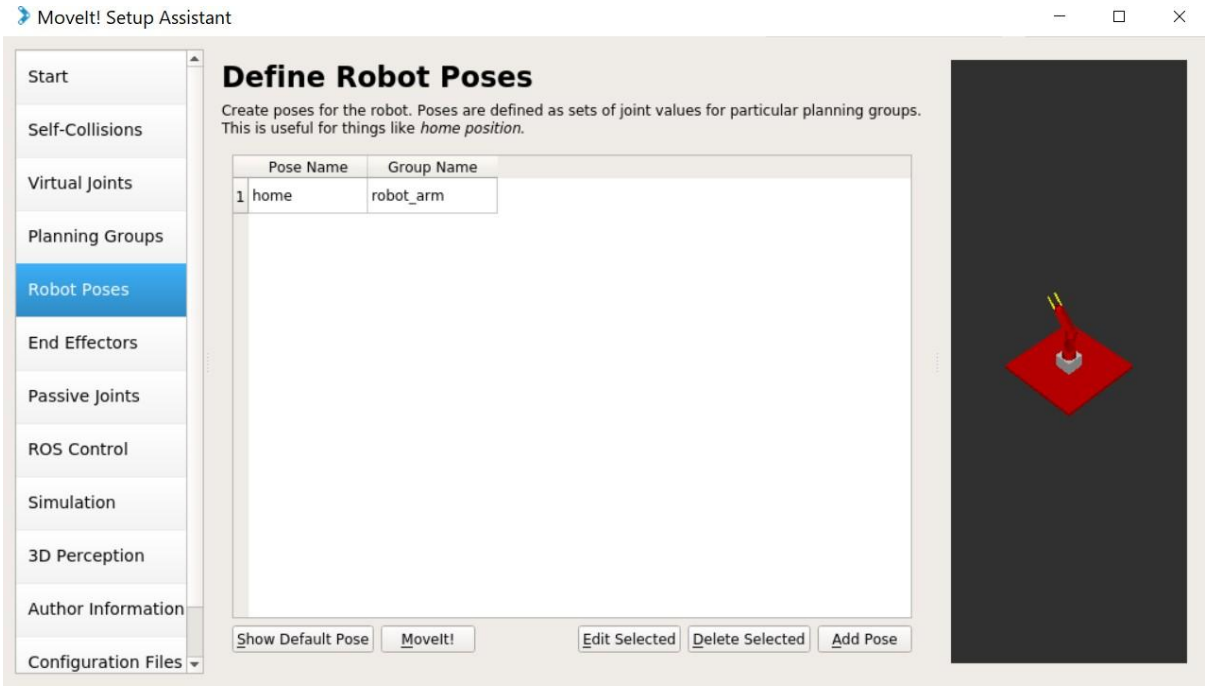


Step 5: Add Robot Poses

The Setup Assistant allows you to add certain fixed poses into the configuration. This helps if, for example, you want to define a certain position of the robot as a Home position.

- Click on the Robot Poses pane.
- Click Add Pose. Choose a name for the pose. The robot will be in its Default position where the joint values are set to the mid-range of the allowed joint value range. Move the individual joints around until you are happy and then Save the pose. Note how poses are associated with particular groups. You can save individual poses for each group.
- IMPORTANT TIP: Try to move all the joints around. If there is something wrong with the joint limits in your URDF, you should be able to see it immediately here.

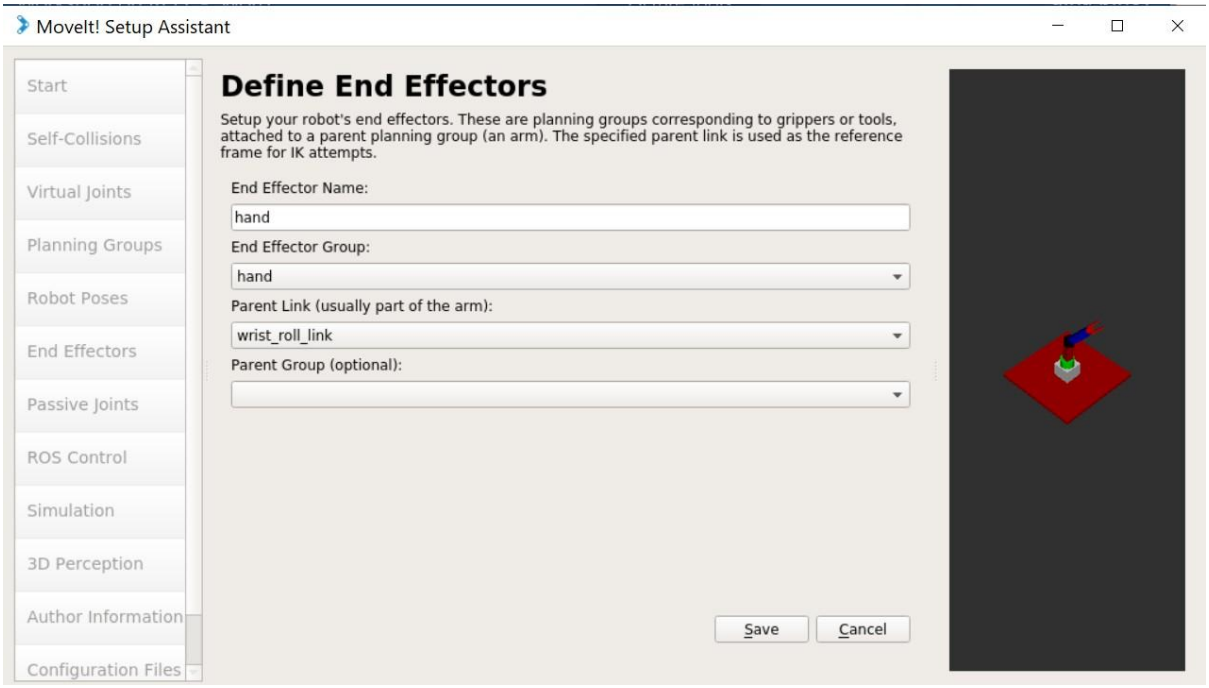




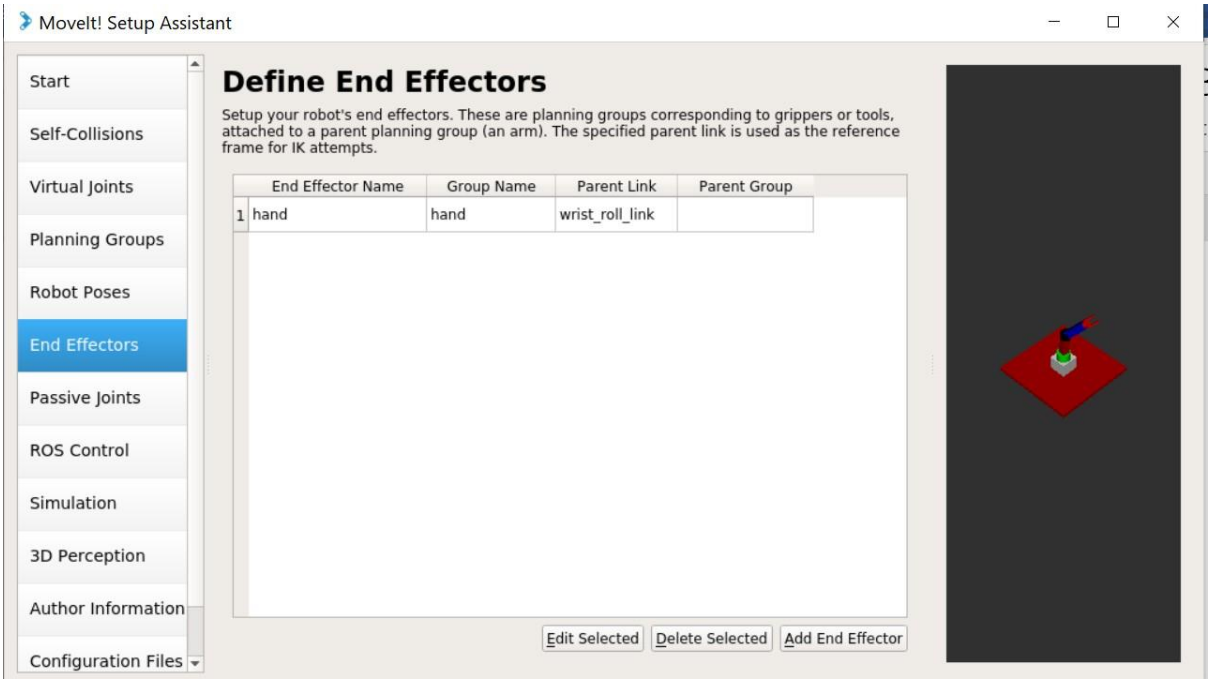
We have already added the gripper. Now, we will designate this group as a special group: end effectors. Designating this group as end effectors allows some special operations to happen on them internally.

- Click on the End Effectors pane.
- Click Add End Effector.
- Choose hand as the End Effector Name for the gripper.

- Select hand as the End Effector Group.
- Select last link of arm as the Parent Link for this end-effector.
- Leave Parent Group blank.



Click on Save



Step 7: Add Passive Joints

The passive joints tab is meant to allow specification of any passive joints that might exist in a robot. These are joints that are unactuated on a robot (e.g. passive casters.) This tells the planners that they cannot (kinematically) plan for these joints because they can't be directly controlled. If the robot does not have any passive joints this step can be skipped.

Step 8: 3D Perception

The 3D Perception tab is meant to set the parameters of the YAML configuration file for configuring the 3D sensors sensors_3d.yaml.

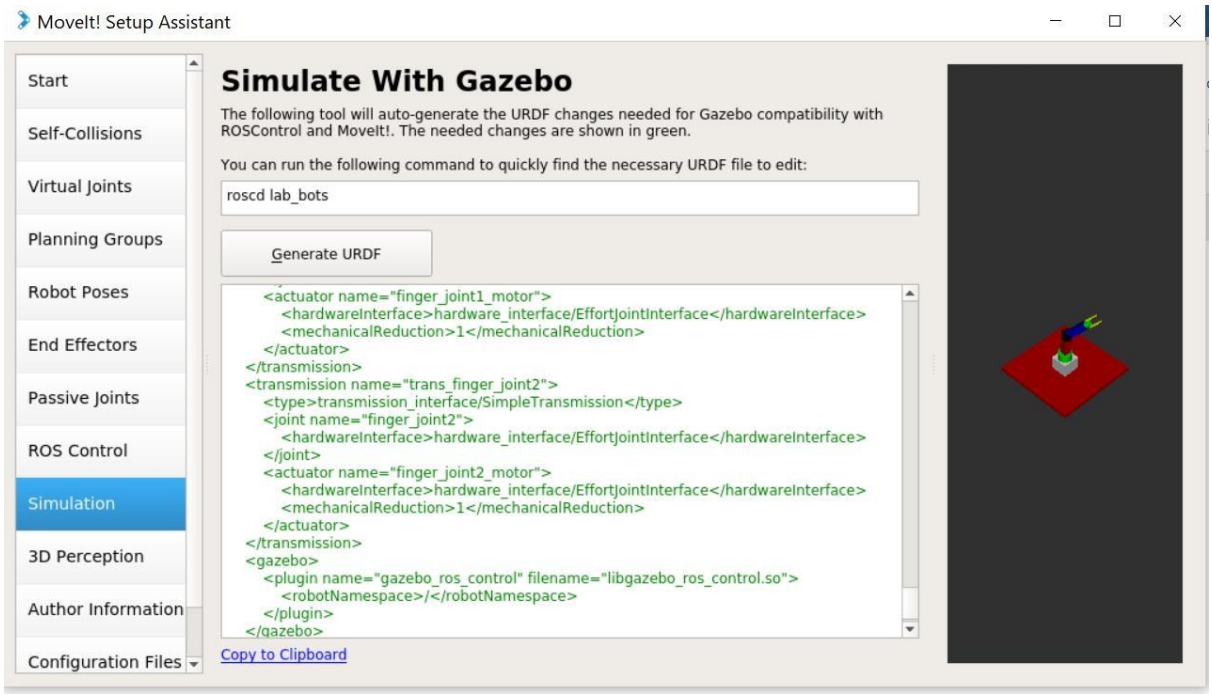
e.g. point_cloud parameters:

For more details about those parameters please see perception pipeline tutorial.

In case of sensors_3d.yaml was not needed, choose None.

Step 9: Gazebo Simulation

The Simulation tab can be used to help you simulate your robot with Gazebo by generating a new Gazebo compatible urdf if needed.

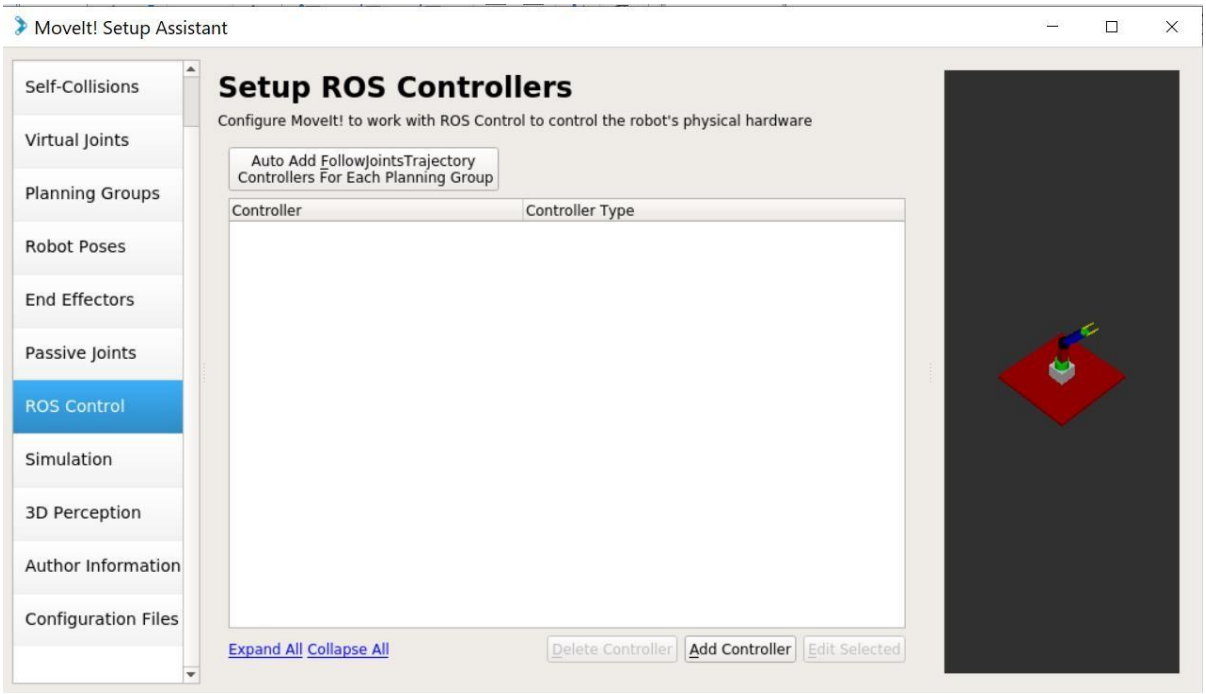


Step 10: ROS Control

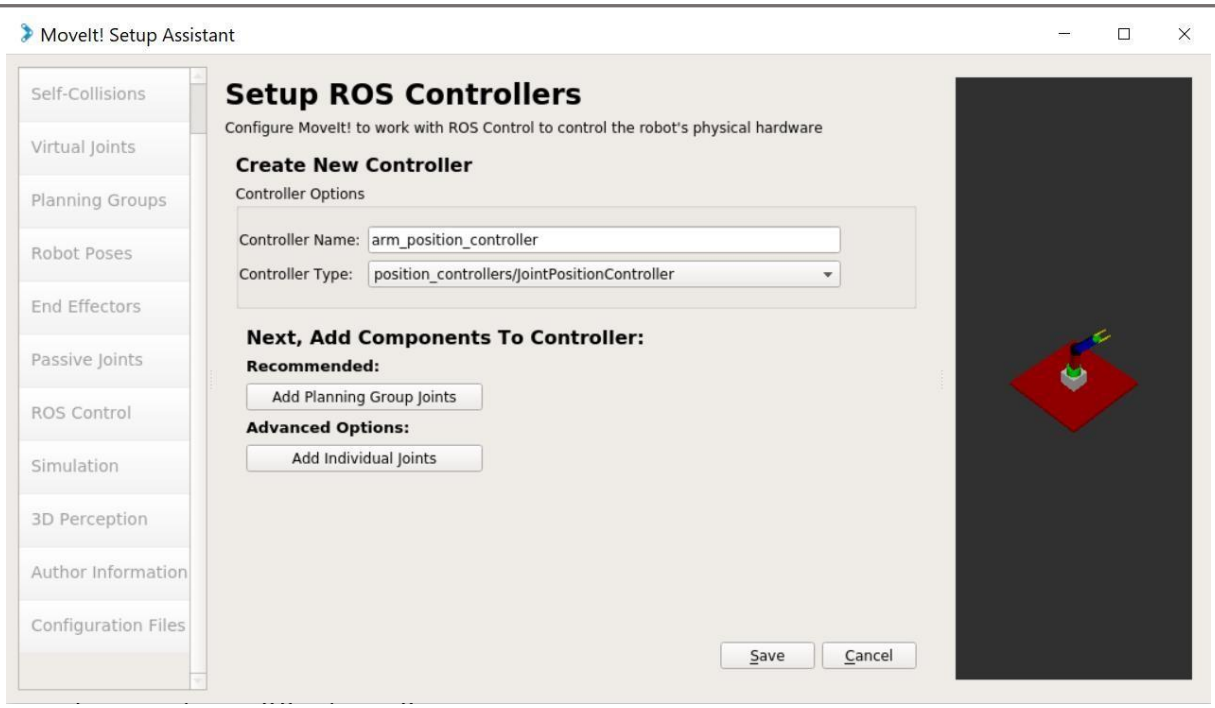
ROS Control is a set of packages that include controller interfaces, controller managers, transmissions and hardware_interfaces, for more details please look at ros_control documentation

ROS Control tab can be used to auto generate simulated controllers to actuate the joints of your robot. This will allow us to provide the correct ROS interfaces MoveIt!.

- Click on the ROS Control pane selector.

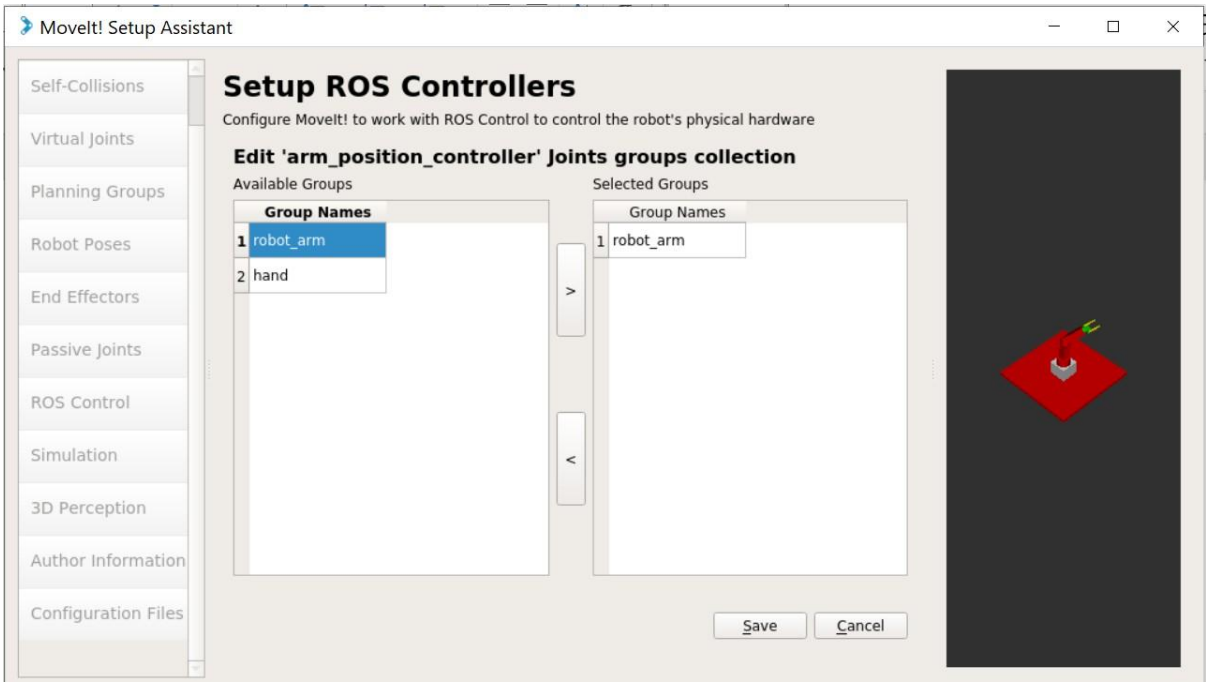


- Click on Add Controller and you should see the following screen:
- We will first add robot arm position controller
- Enter Controller Name as arm_position_controller
- Choose position_controllers/JointPositionController as the controller type



- Next you have to choose this controller joints, you can add joints individually or add all the joints in a planning group all together.

- Now, click on Add Planning Group Joints.

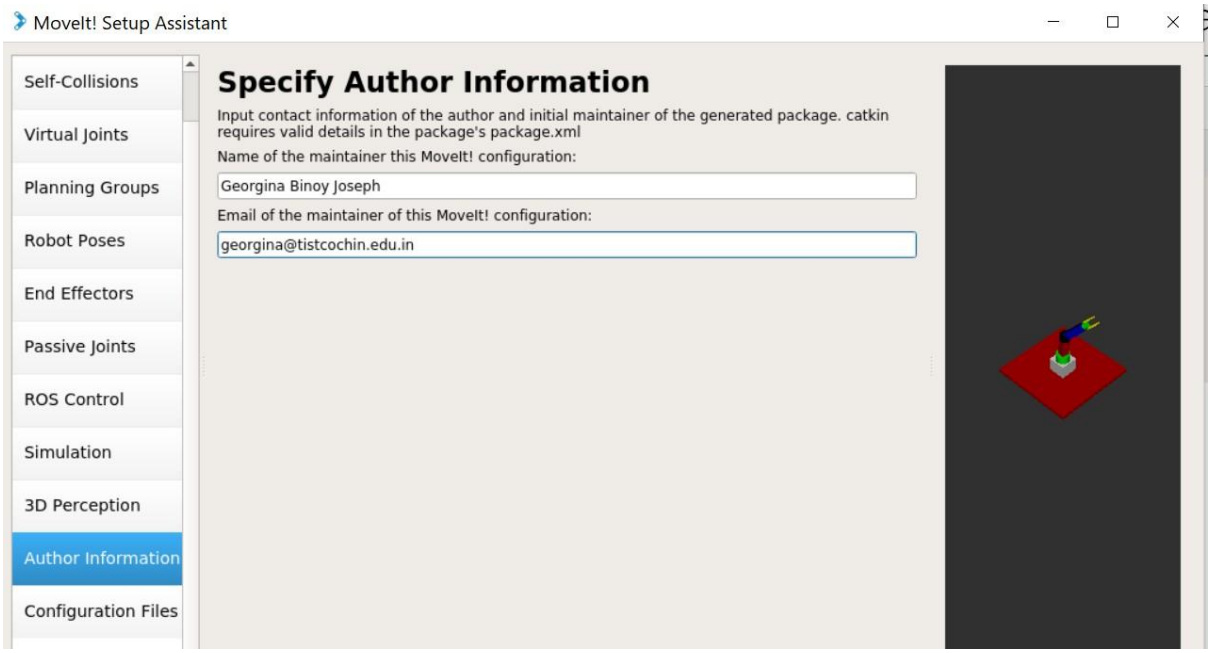


Click Save to save the selected controller.

Step 11: Add Author Information

Catkin requires author information for publishing purposes

- Click on the Author Information pane.
- Enter your name and email address.



Step 12: Generate Configuration Files

The last step - generating all the configuration files that you will need to start using MoveIt!

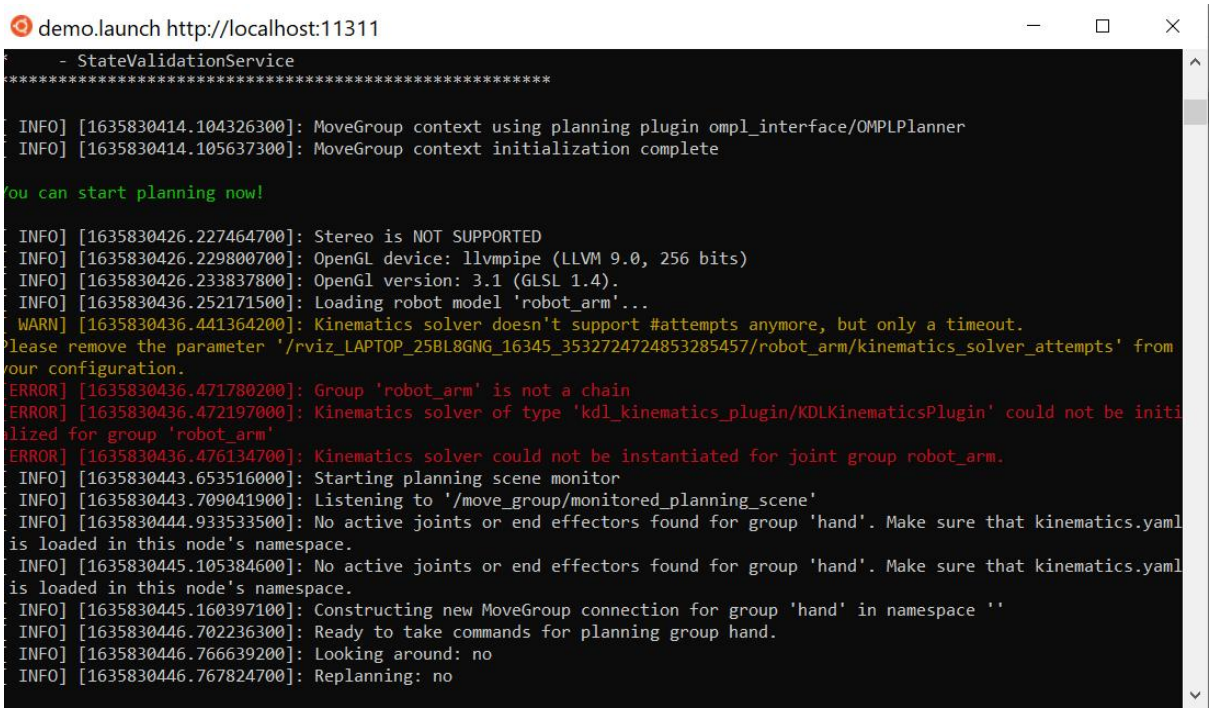
- Click on the *Configuration Files* pane. Choose a location and name for the ROS package that will be generated containing your new set of configuration files. Click browse, select a good location (for example, your home directory or /home/gina/catkin_ws/src/) , click **Create New Folder**, call it "robot_arm_moveit_config", and click **Choose**. This package does not have to be within your ROS package path. All generated files will go directly into the directory you have chosen.
- Click on the *Generate Package* button. The Setup Assistant will now generate and write a set of launch and config files into the directory of your choosing. All the generated files will appear in the Generated Files/Folders tab and you can click on each of them for a description of what they contain.
- Exit Setup assistant

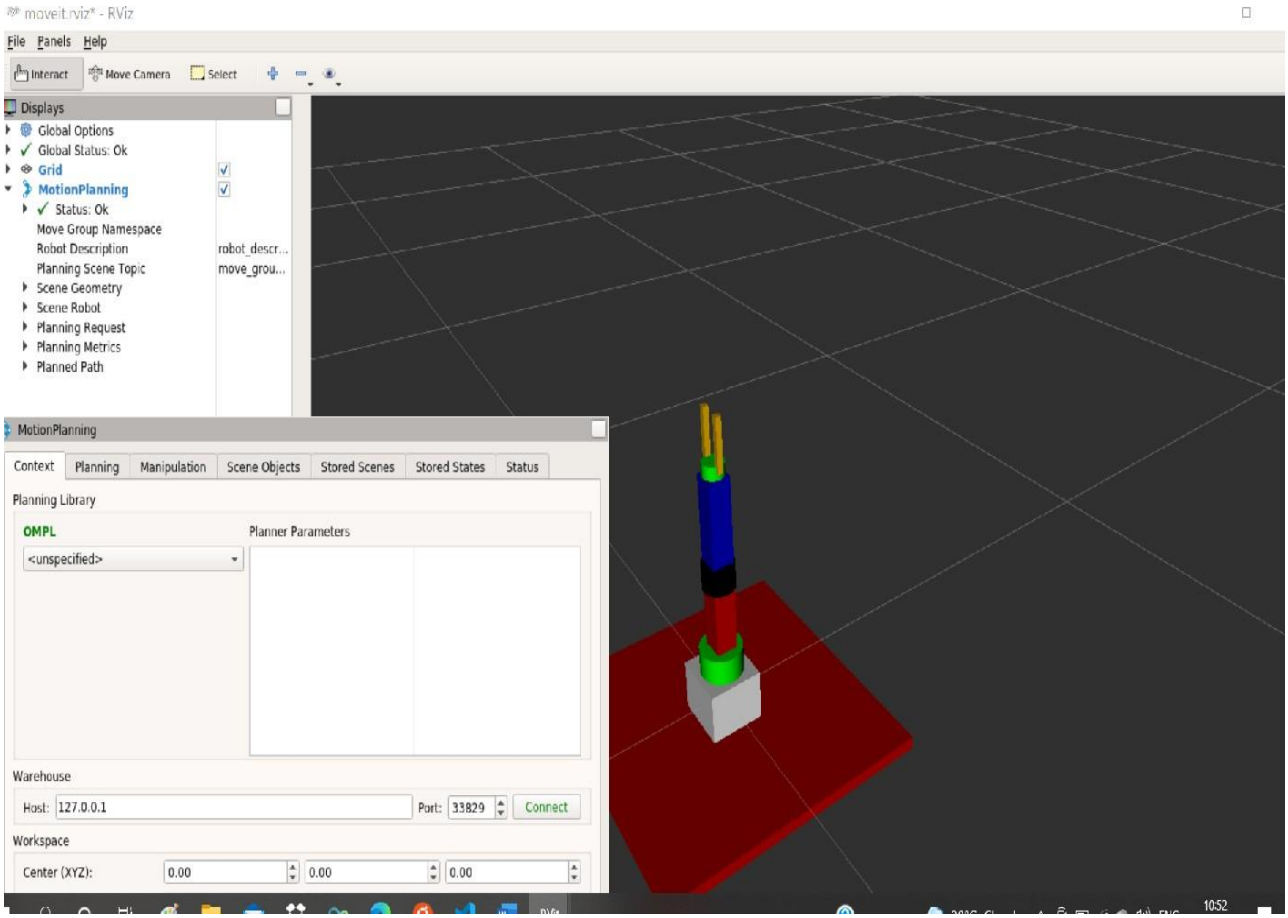
cd ~/catkin_ws

catkin_make

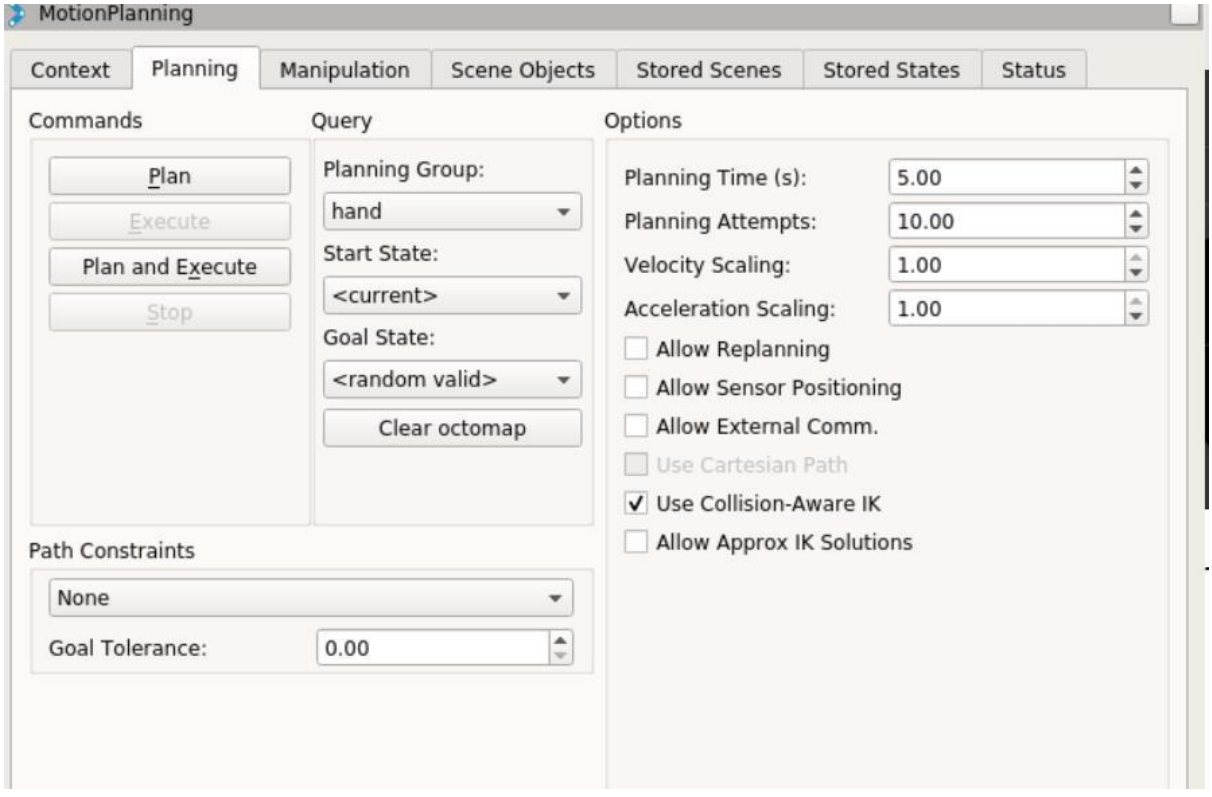
cd src/robot_arm_moveit_config/launch

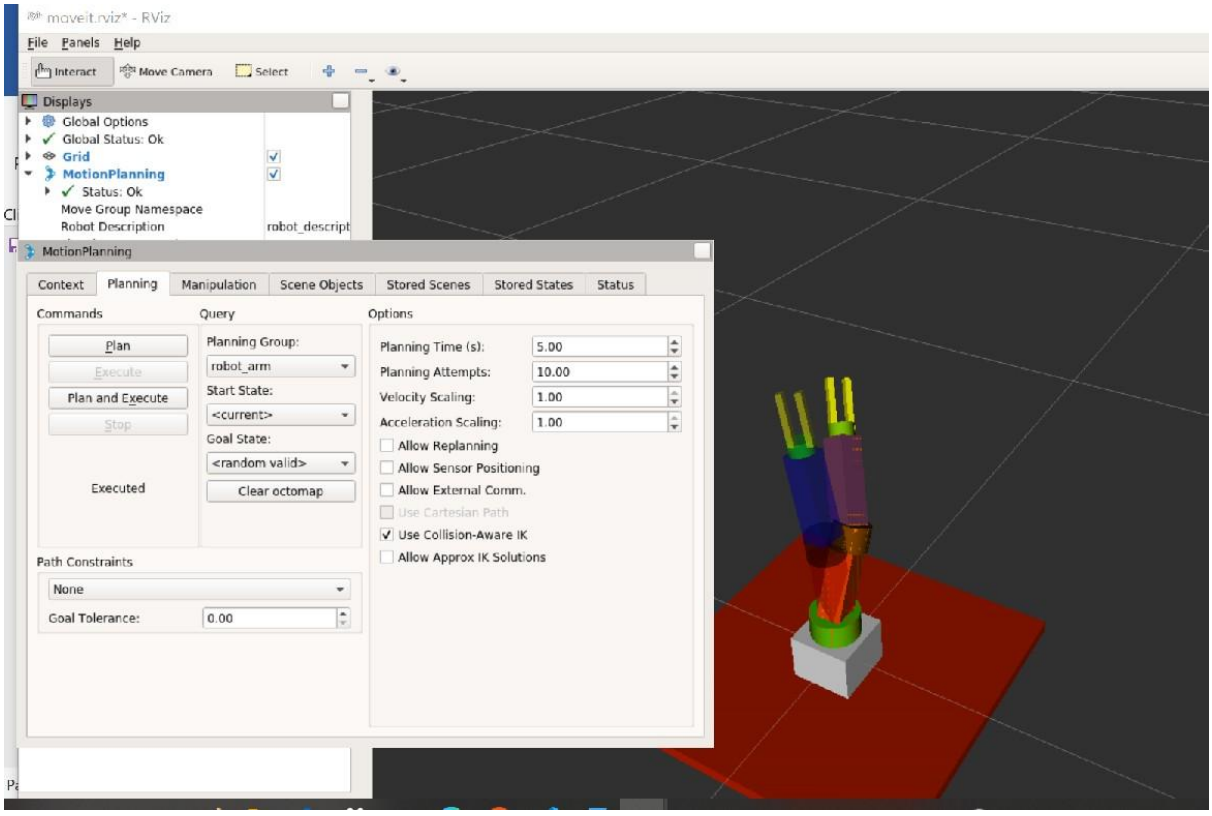
roslaunch demo.launch

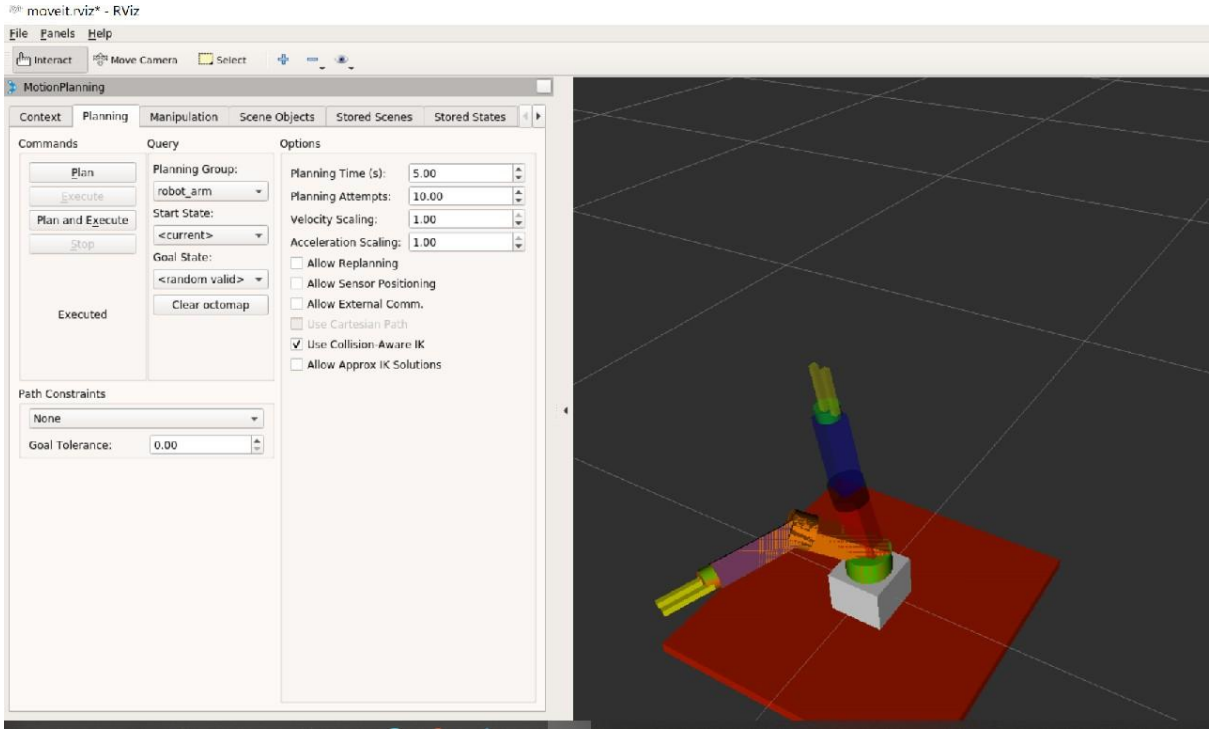




Select Goal State as Random valid, the click Plan and Execute




moveit.rviz* - RViz
File Panels Help
Interact Move Camera Select
Displays
Global Options
Global Status: Ok
Grid
MotionPlanning
Status: Ok
Move Group Namespace
Robot Description
MotionPlanning
Context Planning Manipulation Scene Objects Stored Scenes Stored States Status
Commands
Plan
Execute
Plan and Execute
Stop
Executed
Query
Planning Group:
robot_arm
Start State:
<current>
Goal State:
<random valid>
Clear octomap
Options
Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
Allow Replanning
Allow Sensor Positioning
Allow External Comm.
Use Cartesian Path
Use Collision-Aware IK
Allow Approx IK Solutions
Path Constraints
None
Goal Tolerance: 0.00


moveit.rviz* - RViz
File Panels Help
Interact Move Camera Select
MotionPlanning
Context Planning Manipulation Scene Objects Stored Scenes Stored States
Commands
Plan
Execute
Plan and Execute
Stop
Executed
Query
Planning Group:
robot_arm
Start State:
<current>
Goal State:
<random valid>
Clear octomap
Options
Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
Allow Replanning
Allow Sensor Positioning
Allow External Comm.
Use Cartesian Path
Use Collision-Aware IK
Allow Approx IK Solutions
Path Constraints
None
Goal Tolerance: 0.00

Experiment.10

AIM: Execute SLAM Mapping (Lidar based) using a differentially driven mobile robot

Prerequisite :

1. Installation of turtlebot3

(https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/)

```
$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
ros-noetic-rosserial-python ros-noetic-rosserial-client \
ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
ros-noetic-compressed-image-transport ros-noetic-rqt* ros-noetic-rviz \
ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-markers
$ sudo apt install ros-noetic-turtlebot3-msgs
$ sudo apt install ros-noetic-turtlebot3
```

Install Turtlebot3 Simulation Package

```
cd ~/catkin_ws/src/
$ git clone -b noetic-devel https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin_make
```

2. Installation of GMapping Package

```
sudo apt install ros-noetic-slam-gmapping
```

Procedure:

Step 1:Loading Turtlebot3 with an environment

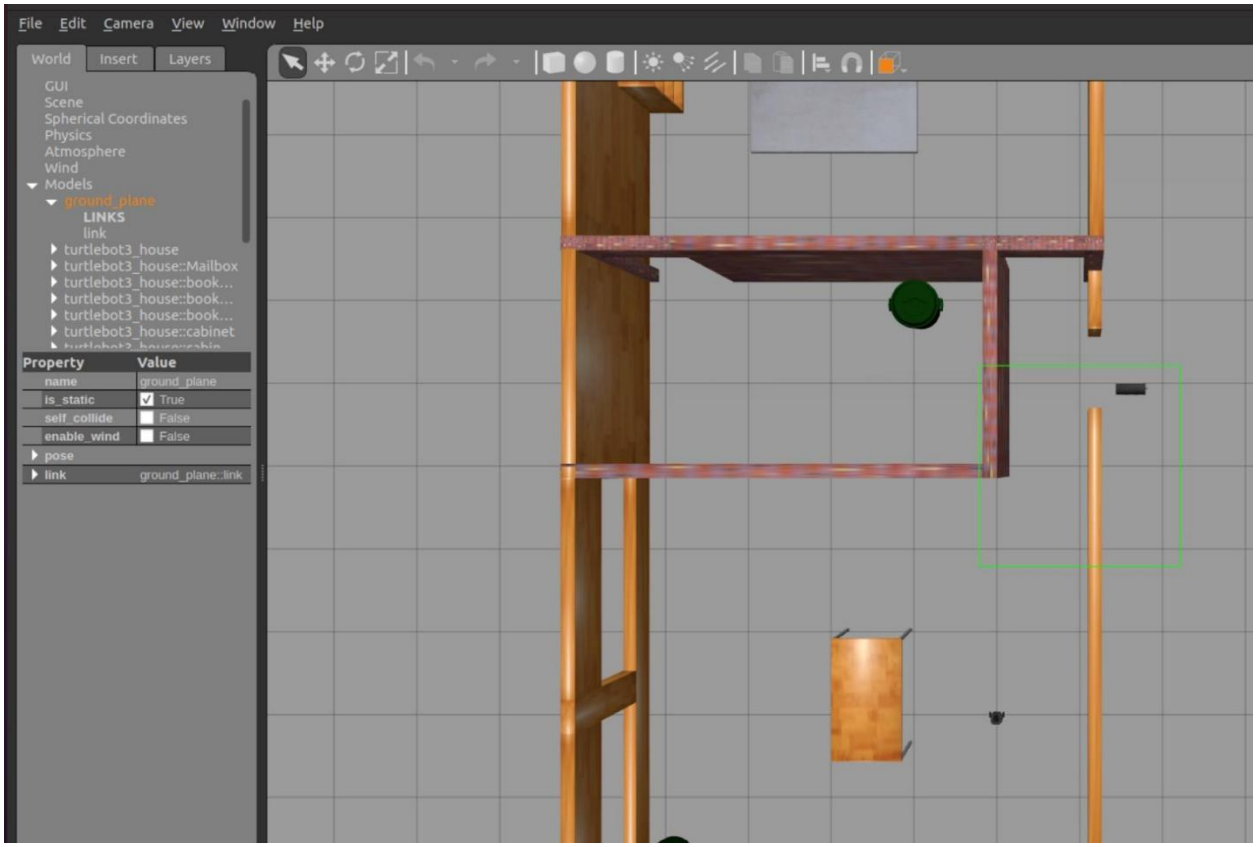
Terminal 1>>

cd catkin_ws/

source devel/setup.bash

export TURTLEBOT3_MODEL=burger

roslaunch turtlebot3_gazebo turtlebot3_house.launch



Step 2:Load Teleopkey for moving turtlebot

Terminal 2>>

cd catkin_ws/

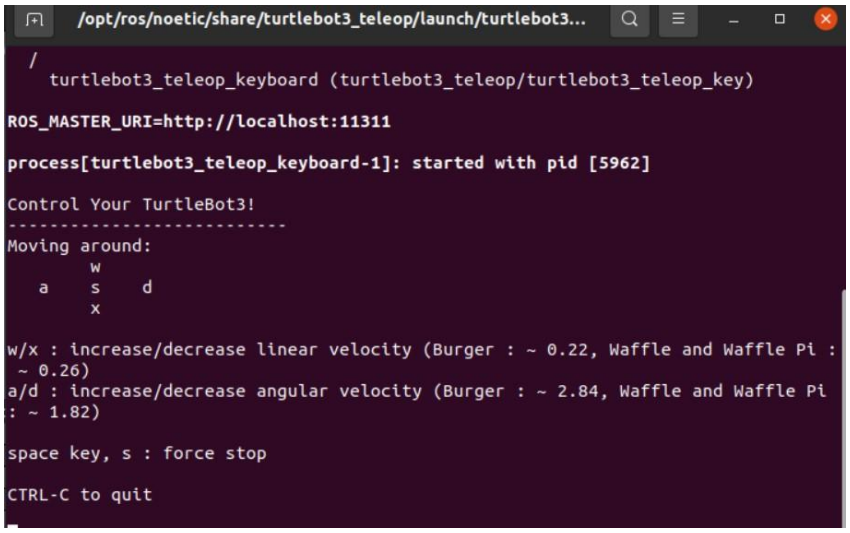
source devel/setup.bash

export TURTLEBOT3_MODEL=burger

roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch (Manual control using keyboard)

OR

roslaunch turtlebot3_gazebo turtlebot3_simulation.launch(Autonomous Navigation)



Step 3:Visualizing Laser Scan Data

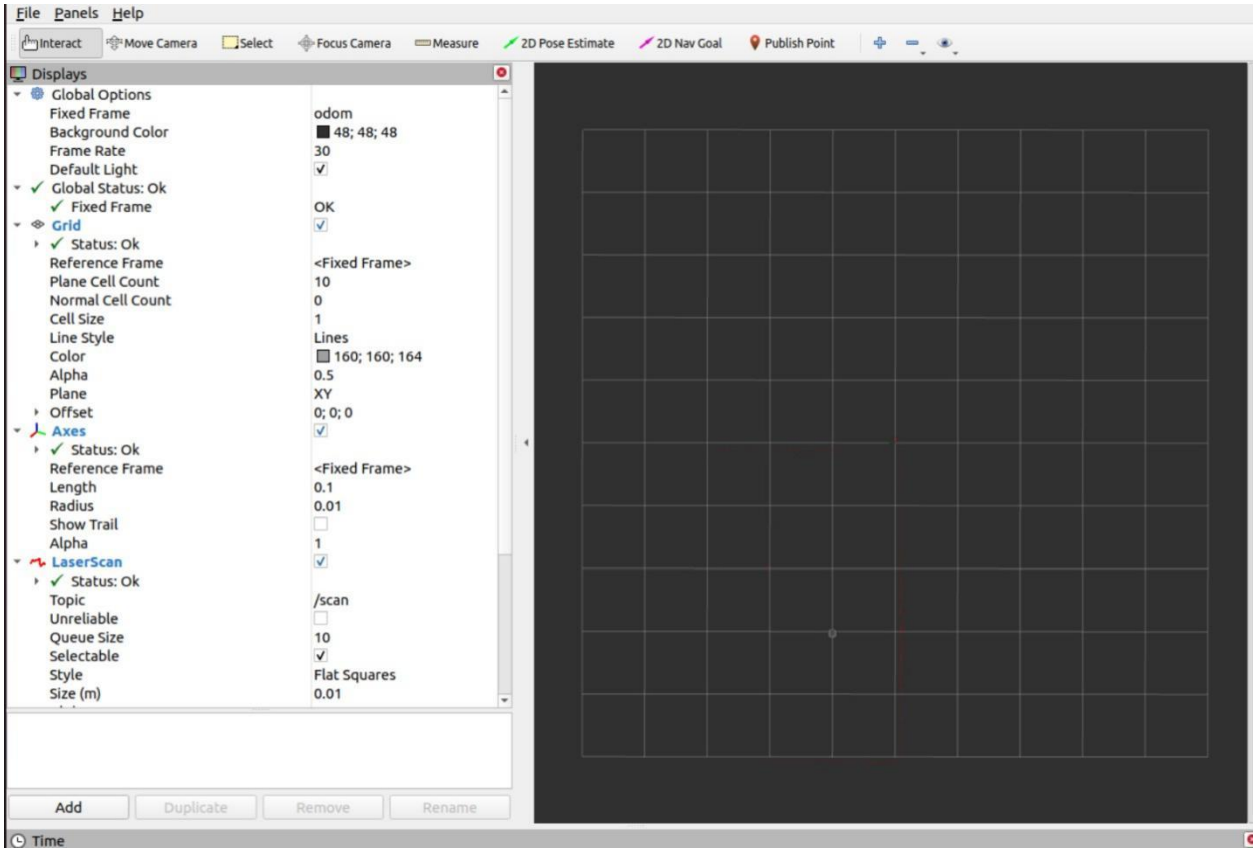
Terminal 3>>

cd catkin_ws/

source devel/setup.bash

export TURTLEBOT3_MODEL=burger

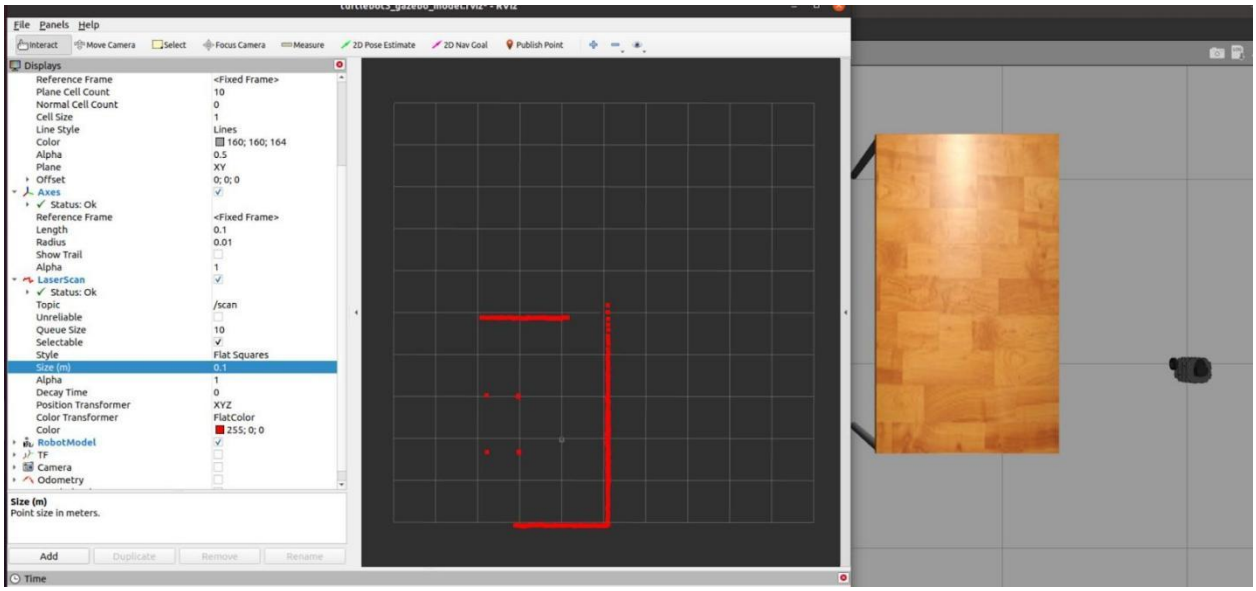
roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch



Increase the laser scan data point size to 0.1

laser scan

size = 0.1



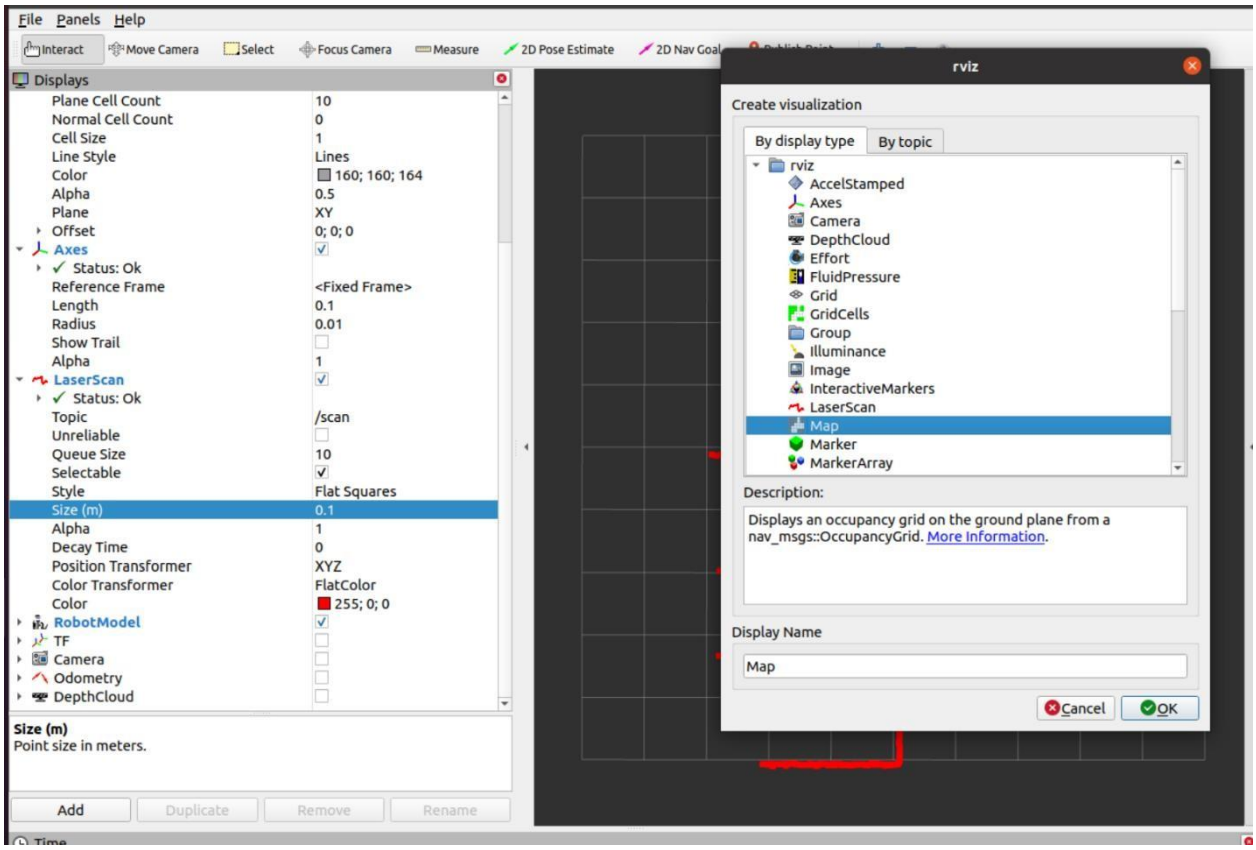
Step 4:Launch GMapping SLAM Package

Terminal 4>>

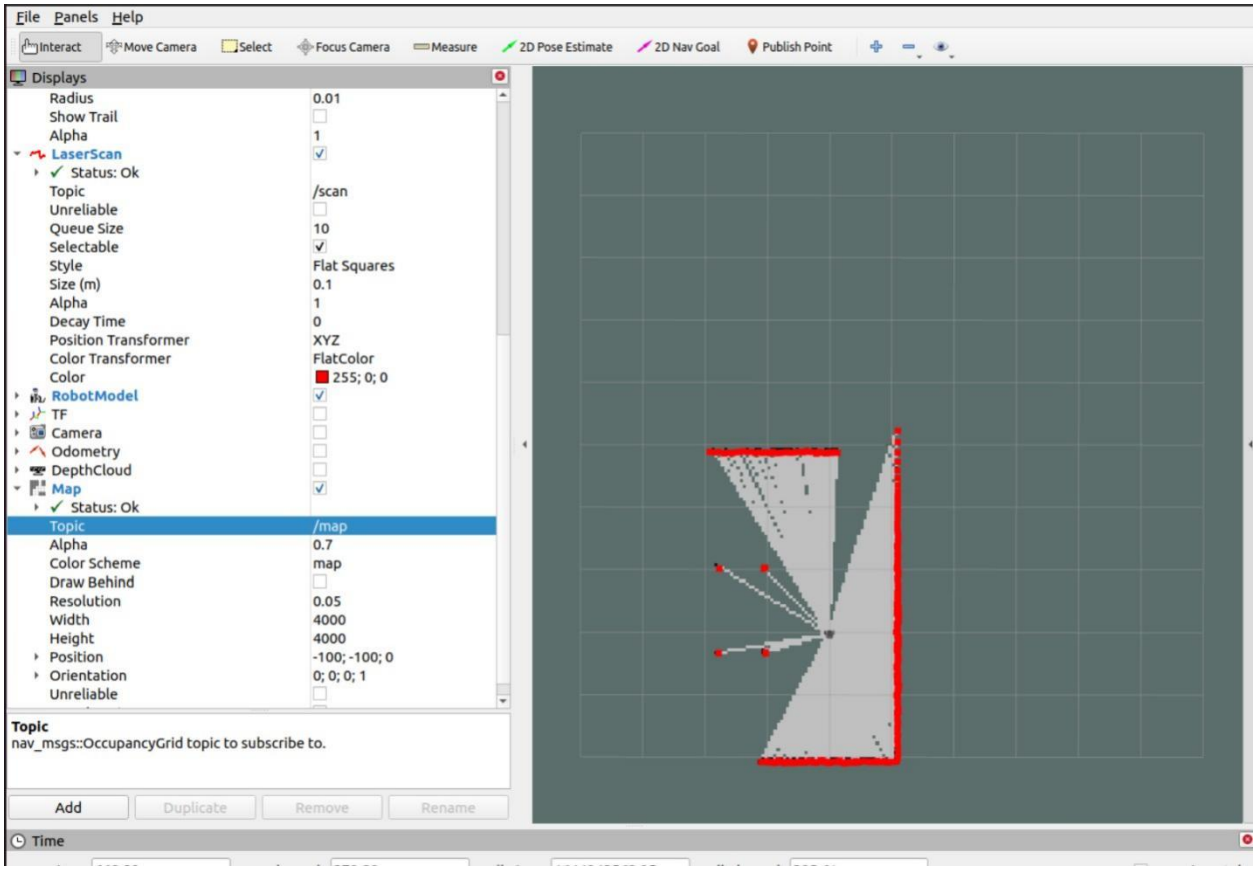
rosrun gmapping slam_gmapping scan:=/scan

In Rviz add ‘Map’ for visualizing Map

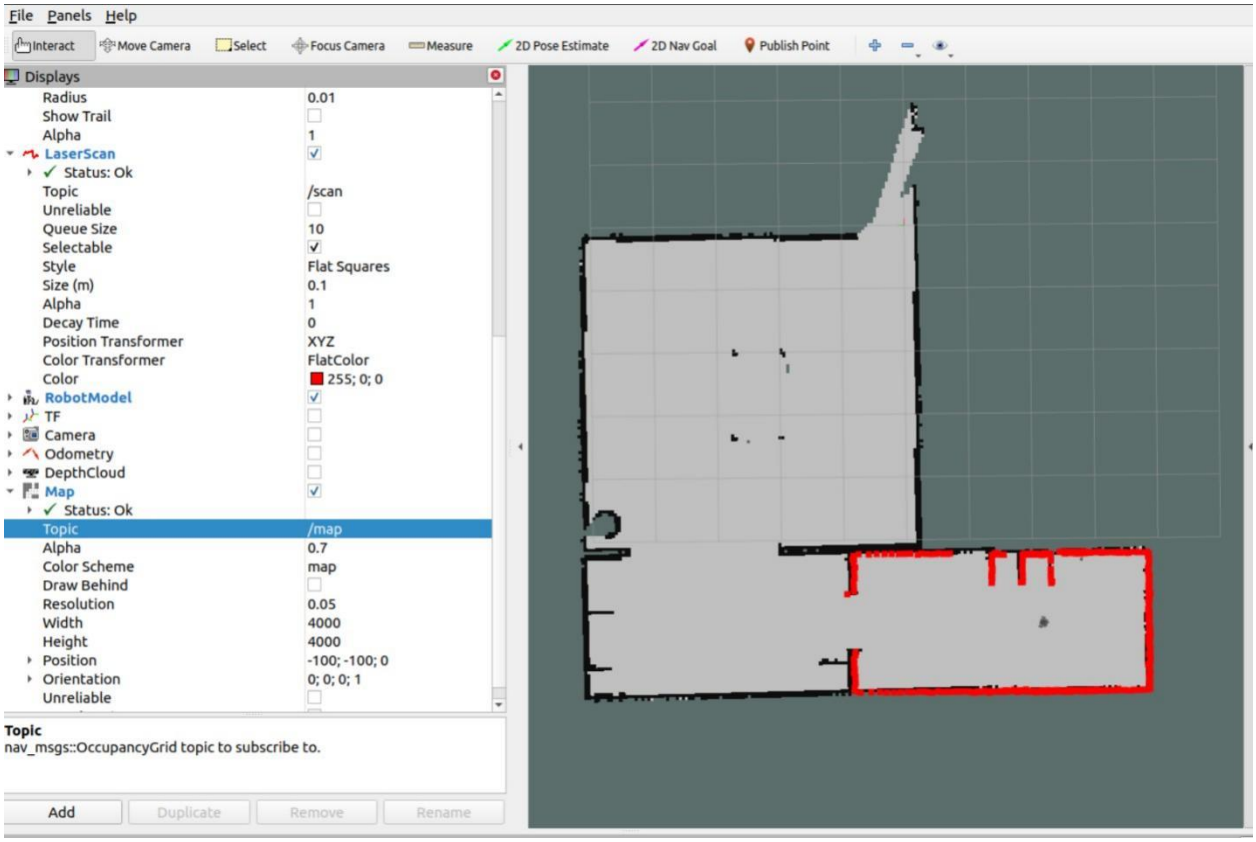
Add Map



Select Topic: /map



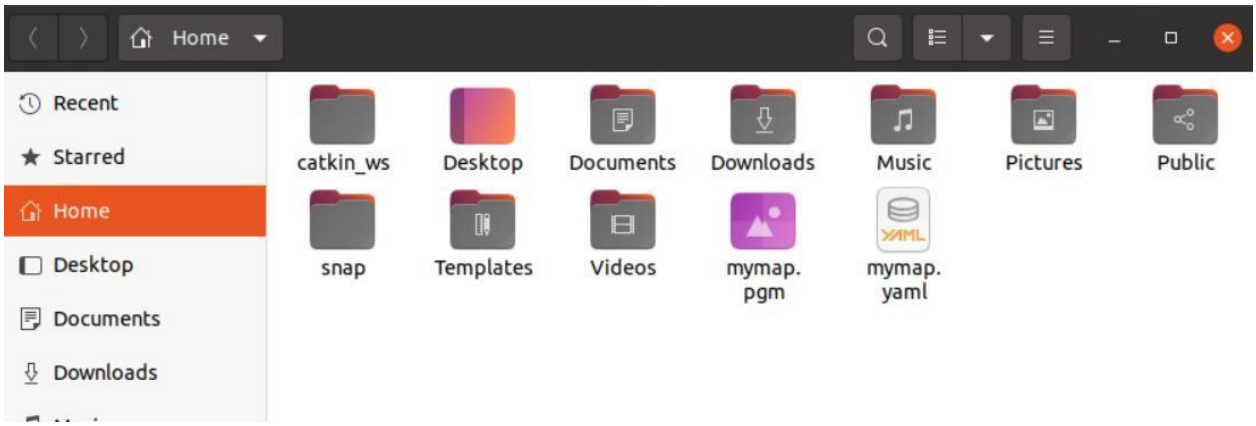
Step 5: Drive turtlebot in the environment to map the surroundings using the teleopkey



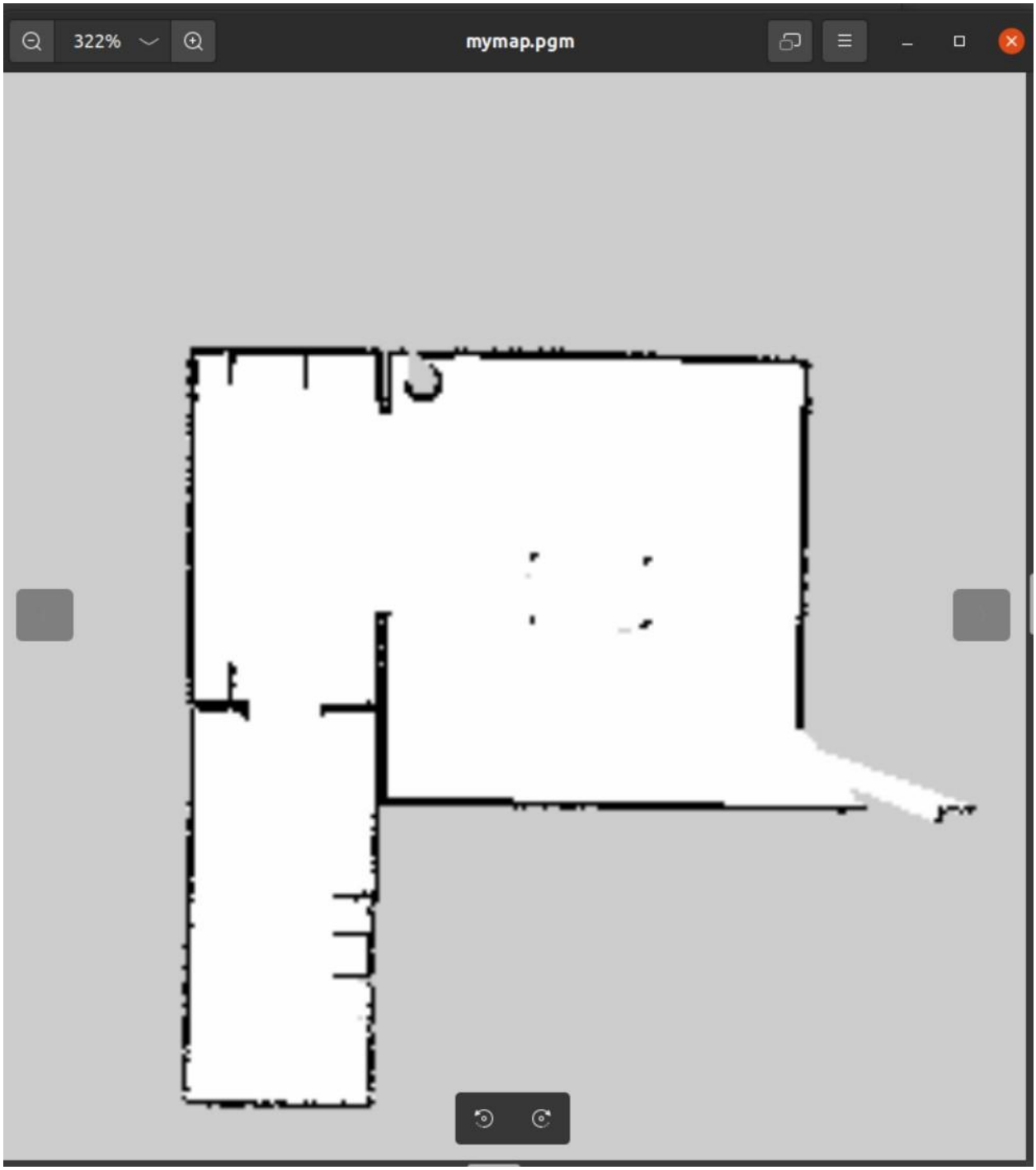
Step 5:Saving the Map using Map server

Terminal 5>>

rosrun map_server map_saver -f mymap



Output:


322%
mymap.pgm

PART B

Experiment.11

AIM: Familiarise ROS Serial Arduino for hardware interface.

The Arduino and Arduino IDE are great tools for quickly and easily programming hardware. Using the rosserial_arduino package, you can use ROS directly with the Arduino IDE. rosserial provides a ROS communication protocol that works over your Arduino's UART. It allows your Arduino to be a full-fledged ROS node which can directly publish and subscribe to ROS messages, publish TF transforms, and get the ROS system time.

Install Arduino IDE on Ubuntu 20.04

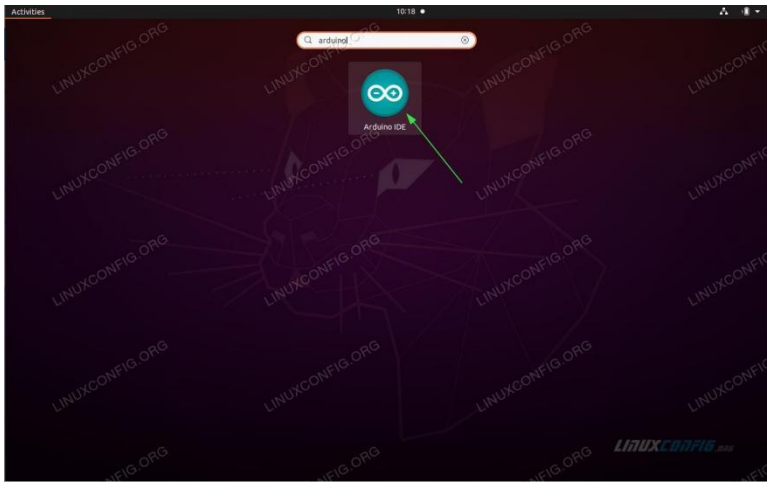
STEP1:

Open up a terminal window and use the **snap** command to install Arduino IDE on your Ubuntu 20.04 desktop. To do so execute the command below:

sudo snap install arduino

STEP2:

The Arduino IDE can now be started from your Activities top left menu.



Alternatively, you can start the Arduino IDE from the Linux command line be executing:

arduino

STEP3:

Installing Arduino on the ROS workstation

You can install rosserial for Arduino by running:

sudo apt-get install ros-noetic-rosserial-Arduino

sudo apt-get install ros-noetic-rosserial

STEP4:

Installing from Source onto the ROS workstation

To clone rosserial from the github repository, generate the rosserial_msgs needed for communication, and make the library files in the <ws>/devel/lib directory.

mkdir name of your workspace

cd name of your workspace

mkdir src

cd src

git clone https://github.com/ros-drivers/rosserial.git

cd name of your workspace

catkin_make

catkin_make install

STEP5:

Install ros_lib into the Arduino Environment

The given steps will create the ros_lib folder that the Arduino build environment needs to enable Arduino programs to interact with ROS.

In the steps below, <sketchbook> is the directory where the Linux Arduino environment saves your sketches. Typically this is a directory called sketchbook or Arduino in your home directory. e.g cd ~/Arduino/libraries.

Copy the path in the terminal

rm -rf ros_lib

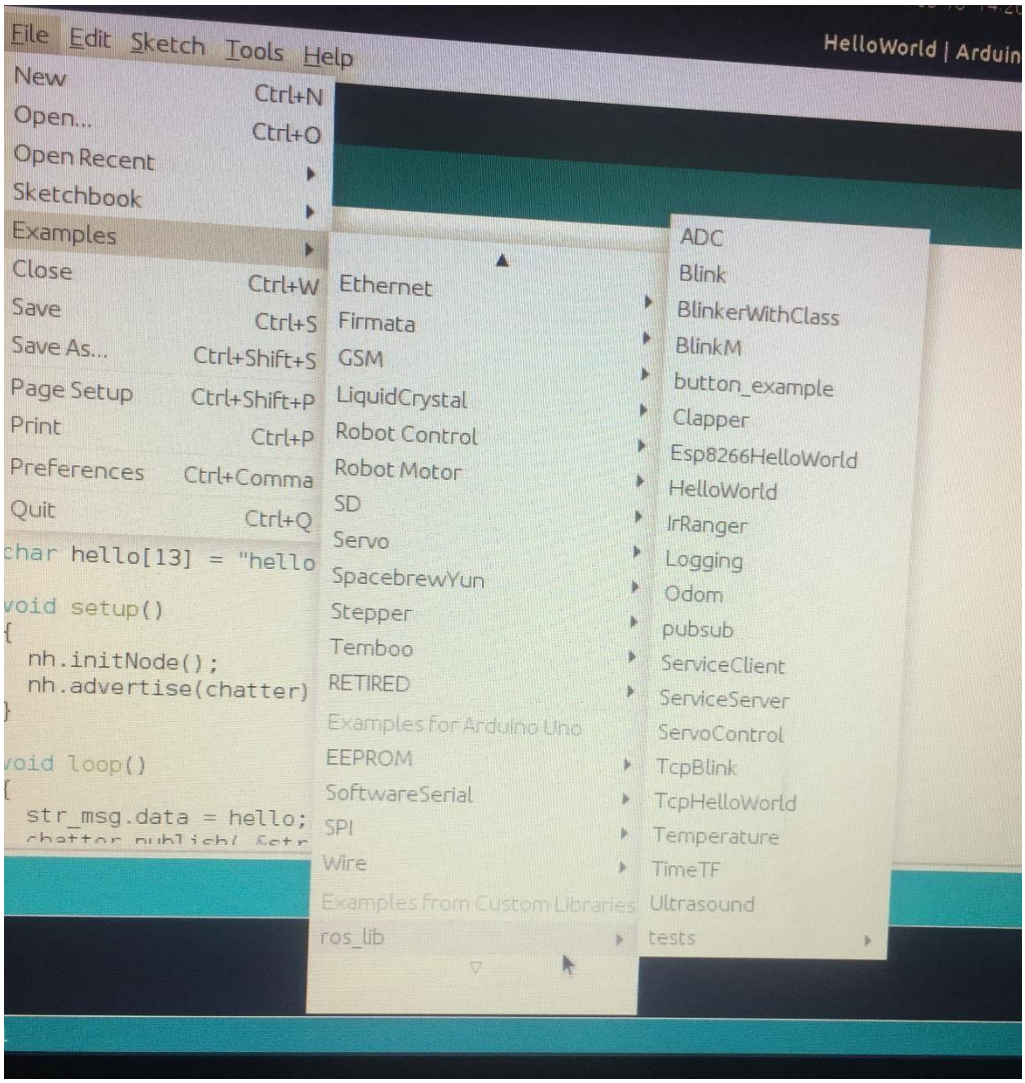
rosrun rosserial_arduino make_libraries.py .

STEP6:

Currently you can install the Arduino libaries directly in the Arduino IDE. Just open the Library Manager from the IDE menu in Sketch -> Include Library -> Manage Library. Then search for "rosserial". This is useful if you need to work on an Arduino sketch but don't want to setup a full ROS workstation.

STEP7:

You can see the roslib in Arduino IDE- FILE -> EXAMPLES -> roslib



PROGRAM

A. Hello World: Creating a Publisher

STEP1:

Copy the code in arduino IDE:

```
/*
 * rosserial Publisher Example
 * Prints "hello world!"
 */

#include <ros.h>
#include <std_msgs/String.h>

ros::NodeHandle nh;

std_msgs::String str_msg;
ros::Publisher chatter("chatter", &str_msg);

char hello[13] = "hello world!";

void setup()
{
  nh.initNode();
  nh.advertise(chatter);
}

void loop()
{
  str_msg.data = hello;
  chatter.publish( &str_msg );
  nh.spinOnce();
  delay(1000);
}
```

STEP2: **Compiling the Code**

To compile the code, use the compile function within the Arduino IDE.

STEP3: **Uploading the Code**

To upload the code to your Arduino, use the upload function within the Arduino IDE.

STEP4: **Running the Code**

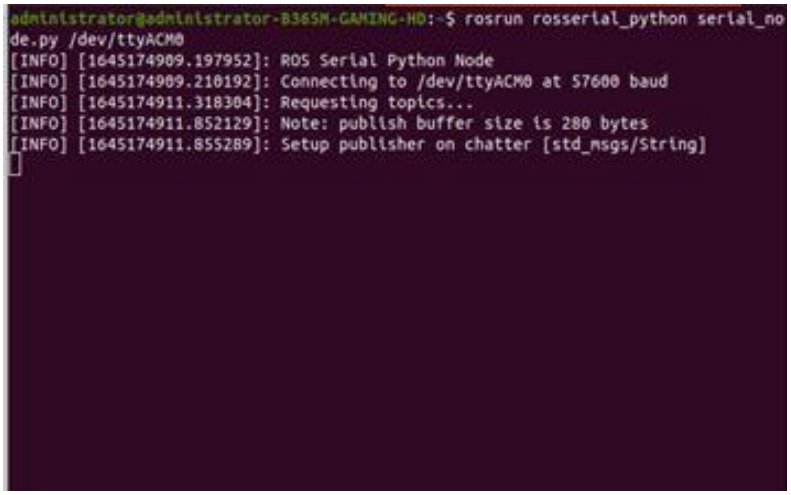
1. launch the roscore in a new terminal window

   roscore

2. Run the rosserial client application that forwards your Arduino messages to the rest of ROS. Make sure to use the correct serial port:

   In a new terminal window

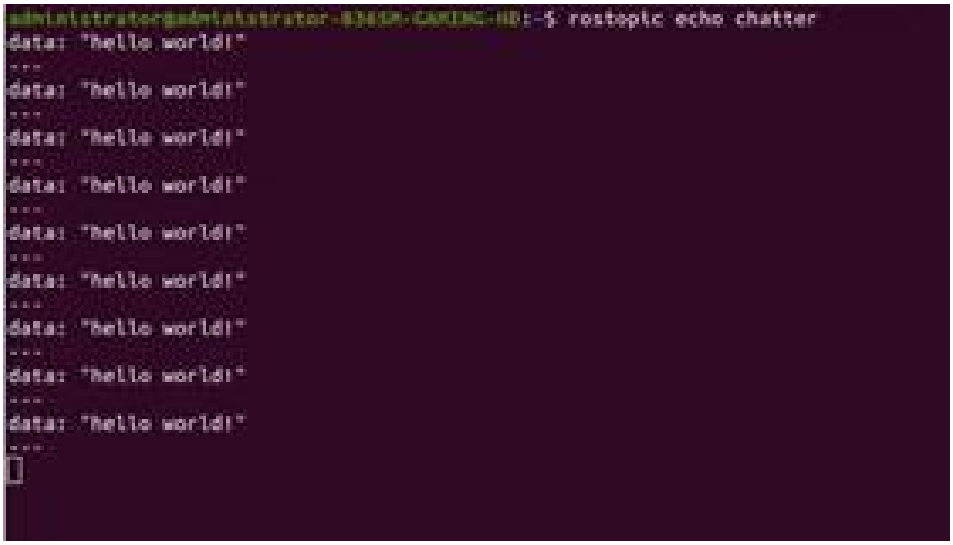
   rosrun rosserial_python serial_node.py /dev/ttyACM0



3. To watch the greetings come in from your Arduino by launching a new terminal window and entering**:**

   In a new terminal window

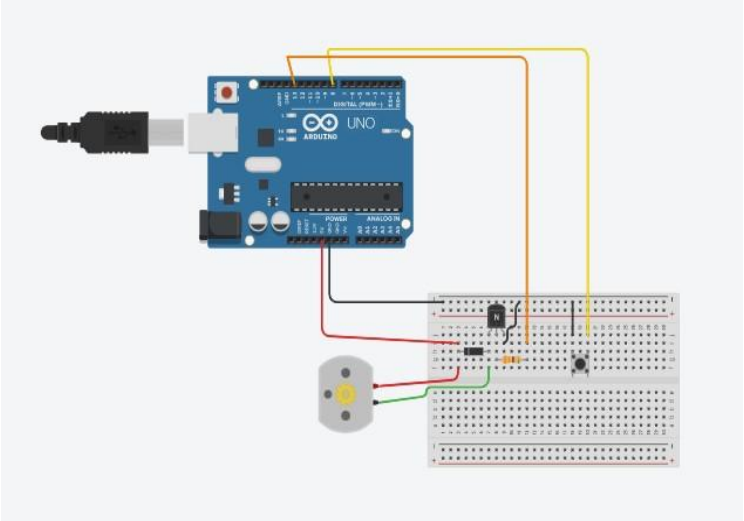
   rostopic echo chatter



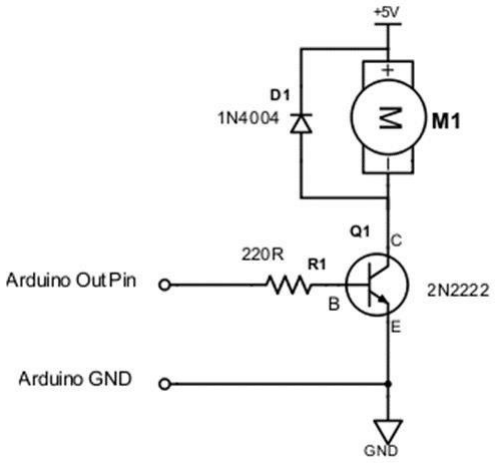
B. To write a program to control the DC motor using BUTTON switch.

STEP1:

Set up the hardware



Driver unit



STEP2:

Copy the code in arduino IDE:

```
#include <ros.h>
#include <std_msgs/Bool.h>


ros::NodeHandle nh;

std_msgs::Bool pushed_msg;
ros::Publisher pub_button("pushed", &pushed_msg);
```

```
const int button_pin = 7;
const int motor_pin = 13;

bool last_reading;
long last_debounce_time=0;
long debounce_delay=50;
bool published = true;

void setup()
{
 nh.initNode();
 nh.advertise(pub_button);

 //initialize an motor output pin
 //and a input pin for our push button
 pinMode(motor_pin, OUTPUT);
 pinMode(button_pin, INPUT);

 //Enable the pullup resistor on the button
 digitalWrite(button_pin, HIGH);


 last_reading = ! digitalRead(button_pin);

}

void loop()
{

 bool reading = ! digitalRead(button_pin);

 if (last_reading!=
    reading){ last_debounce_time
    = millis();published = false;
 }

 //if the button value has not changed for the debounce delay, we know its stable
 if ( !published && (millis() - last_debounce_time) > debounce_delay)
 { digitalWrite(motor_pin, reading);
   pushed_msg.data = reading;
   pub_button.publish(&pushed_msg);
   published = true;
 }

 last_reading = reading;

 nh.spinOnce();
}
```

STEP3:**Compiling the Code**

To compile the code, use the compile function within the Arduino IDE.

STEP4:**Uploading the Code**

To upload the code to your Arduino, use the upload function within the Arduino IDE.

STEP5:**Running the Code**

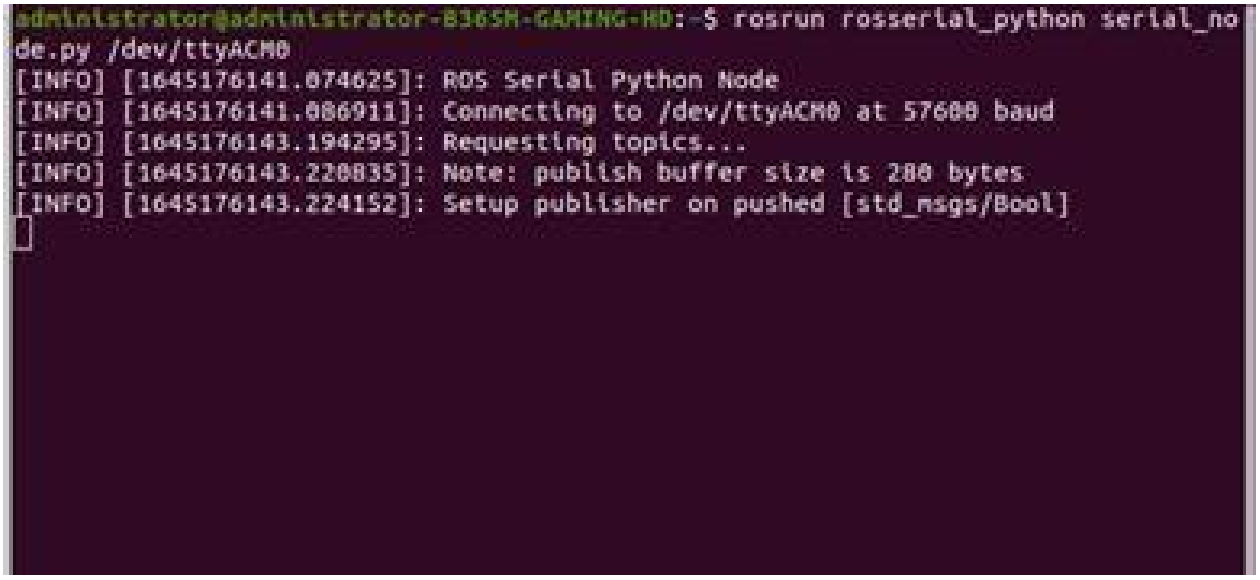
1. launch the roscore in a new terminal window

   roscore

2. Run the rosserial client application that forwards your Arduino messages to the rest of ROS. Make sure to use the correct serial port:

   In a new terminal window

   rosrun rosserial_python serial_node.py /dev/ttyACM0



3. To watch the greetings come in from your Arduino by launching a new terminal window and entering**:**

   In a new terminal window

   rostopic echo pushed


administrator@administrator-B365M-GAMING-HD:~$ rostopic echo pushed
data: True
---
data: False
---
data: True
---
data: False
---
data: True
---
data: False
---
data: True
---
data: False
---
data: True
---
data: False
---
data: True
---
data: False

Additional Experiments-1

1.Creating and Spawning Custom URDF Objects in Simulation

Aim: To create a simple box urdf model using the *box* geometric primitive and spawn it in a simulated empty world.

Steps:

1. Start an empty world simulation

roslaunch gazebo_worlds empty_world.launch

1. Create a Simple Box URDF

```
<robot name="simple_box">
  <link name="my_box">
    <inertial>
      <origin xyz="2 0 0" />
      <mass value="1.0" />
      <inertia  ixx="1.0" ixy="0.0"  ixz="0.0"  iyy="100.0"  iyz="0.0"  izz="1.0"
/>
    </inertial>
    <visual>
      <origin xyz="2 0 1"/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </visual>
    <collision>
      <origin xyz="2 0 1"/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </collision>
  </link>
  <gazebo reference="my_box">
    <material>Gazebo/Blue</material>
  </gazebo>
</robot>
```

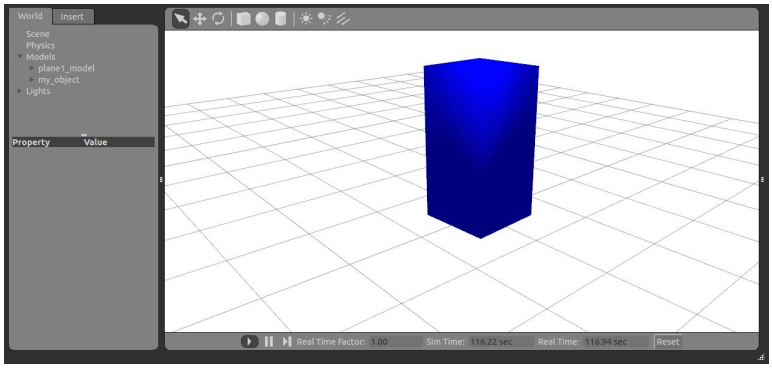
Here the origins of the inertial center, visual and collision geometry centers are offset in +x by 2m relative to the model origin. The box geometry primitive is used here for visual and collision geometry, and has sizes 1m wide, 1m deep and 2m tall. Note that visual and collision geometries do not always have to be the same, sometimes you want to use a simpler collision geometry to save collision detection time when running a dynamic simulation. The box inertia is defined as 1kg mass and principal moments of inertia $i_{xx}=i_{zz}=1$ kg*m^2 and $i_{yy}=100$ kg*m^2.

2. Spawn Model in Simulation

To spawn above URDF object at height **z** = 1 meter and assign the name of the model in simulation to

be **my_object**:

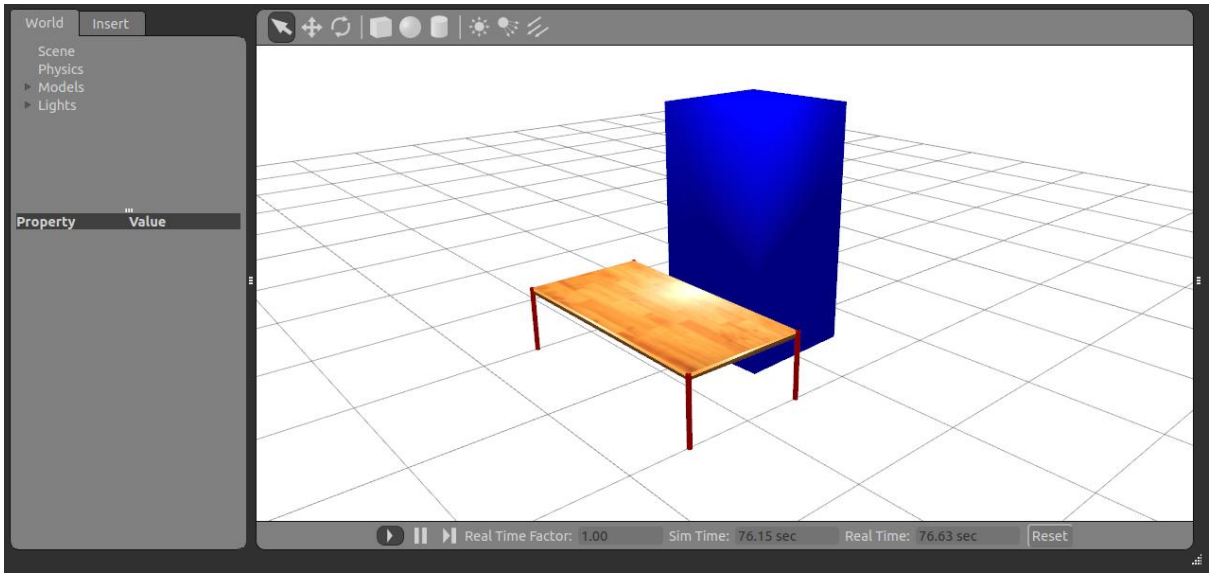
rosrun gazebo spawn_model -file `pwd`/object.urdf -urdf -z 1 -model my_object



To spawn a desk by typing

roslaunch gazebo_worlds table.launch

This should create a table in the simulator GUI (you might have to zoom out and mouse around to find it), then terminate.



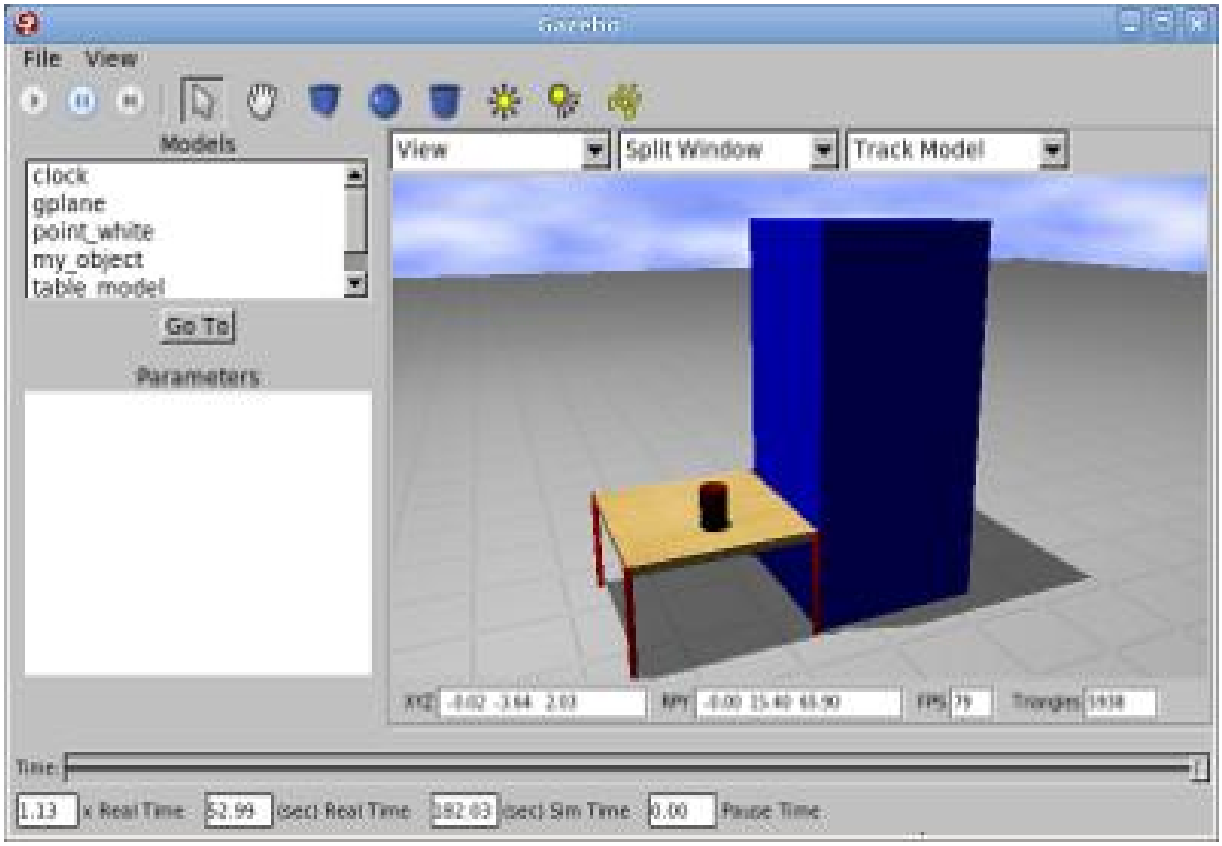

```
<launch>
  <!-- send table urdf to param server -->
  <param name="table_description" command="$(find xacro)/xacro.py $(find gazebo_wo
rlds)/objects/table.urdf.xacro" />

  <!-- push table_description to factory and spawn robot in gazebo -->
  <node name="spawn_table" pkg="gazebo" type="spawn_model" args="-urdf -param tabl
e_description -z 0.01 -model table_model" respawn="false" output="screen" />
</launch>
```

The launch file loads table.urdf.xacro urdf XML file onto the ROS parameter server (after first passing it through the xacro preprocessor). Error: No code_block found

Then calls spawn_model node to spawn the model in simulation. Error: No code_block found The spawn_table node gets the XML string from the parameter server and passes it on to Gazebo to spawn the object in the simulated world.

To spawn a coffee cup on the table

roslaunch gazebo_worlds coffee_cup.launch

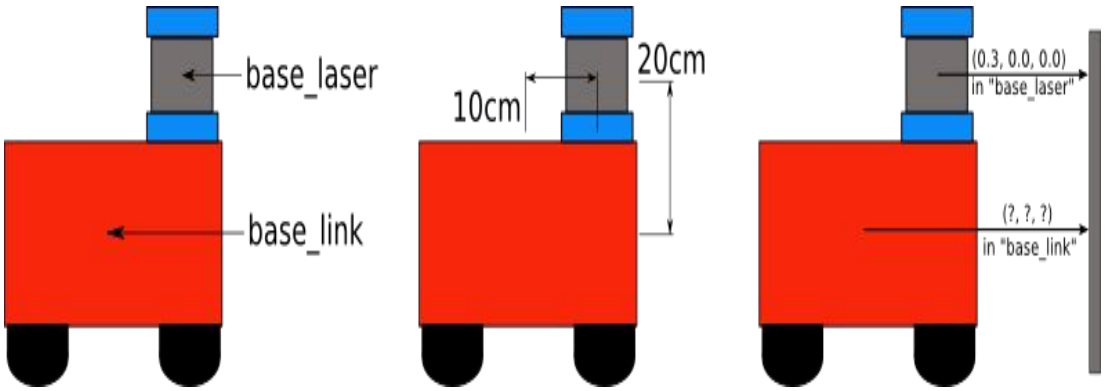


RESULT:

Additional Experiment-2

2.Setting up your robot using tf

Many ROS packages require the transform tree of a robot to be published using the tf software library. At an abstract level, a transform tree defines offsets in terms of both translation and rotation between different coordinate frames. To make this more concrete, consider the example of a simple robot that has a mobile base with a single laser mounted on top of it. In referring to the robot let's define two coordinate frames: one corresponding to the center point of the base of the robot and one for the center point of the laser that is mounted on top of the base. Let's also give them names for easy reference. We'll call the coordinate frame attached to the mobile base "base_link" (for navigation, its important that this be placed at the rotational center of the robot) and we'll call the coordinate frame attached to the laser "base_laser." For frame naming conventions



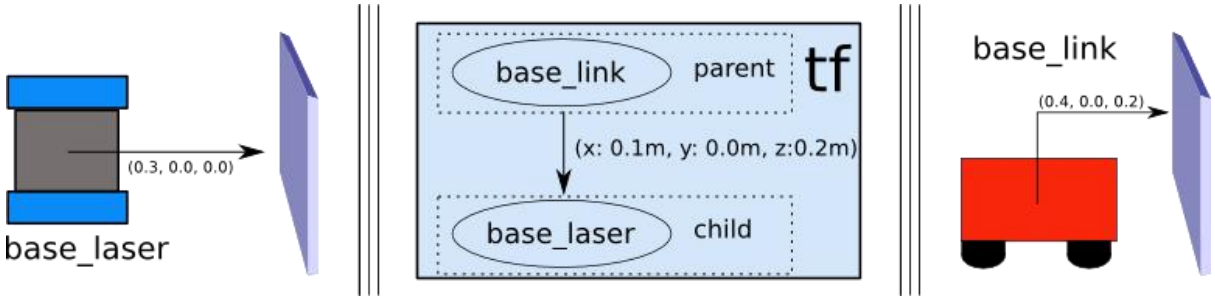
At this point, let's assume that we have some data from the laser in the form of distances from the laser's center point. In other words, we have some data in the "base_laser" coordinate frame. Now suppose we want to take this data and use it to help the mobile base avoid obstacles in the world. To do this successfully, we need a way of transforming the laser scan we've received from the "base_laser" frame to the "base_link" frame. In essence, we need to define a relationship between the "base_laser" and "base_link" coordinate frames.

In defining this relationship, assume we know that the laser is mounted 10cm forward and 20cm above the center point of the mobile base. This gives us a translational offset that relates the "base_link" frame to the "base_laser" frame. Specifically, we know that to get data from the "base_link" frame to the "base_laser" frame we must apply a translation of (x:

0.1m, y: 0.0m, z: 0.2m), and to get data from the "base_laser" frame to the "base_link" frame we must apply the opposite translation (x: -0.1m, y: 0.0m, z: -0.20m).

We could choose to manage this relationship ourselves, meaning storing and applying the appropriate translations between the frames when necessary, but this becomes a real pain as the number of coordinate frames increase. Luckily, however, we don't have to do this work ourselves. Instead we'll define the relationship between "base_link" and "base_laser" once using tf and let it manage the transformation between the two coordinate frames for us.

To define and store the relationship between the "base_link" and "base_laser" frames using tf, we need to add them to a transform tree. Conceptually, each node in the transform tree corresponds to a coordinate frame and each edge corresponds to the transform that needs to be applied to move from the current node to its child. Tf uses a tree structure to guarantee that there is only a single traversal that links any two coordinate frames together, and assumes that all edges in the tree are directed from parent to child nodes.



To create a transform tree for our simple example, we'll create two nodes, one for the "base_link" coordinate frame and one for the "base_laser" coordinate frame. To create the edge between them, we first need to decide which node will be the parent and which will be the child. Remember, this distinction is important because tf assumes that all transforms move from parent to child. Let's choose the "base_link" coordinate frame as the parent because as other pieces/sensors are added to the robot, it will make the most sense for them to relate to the "base_laser" frame by traversing through the "base_link" frame. This means the transform associated with the edge connecting "base_link" and "base_laser" should be (x: 0.1m, y: 0.0m, z: 0.2m). With this transform tree set up, converting the laser scan received in the "base_laser" frame to the "base_link" frame is as simple as making a call to the tf library. Our robot can use this information to reason about laser scans in the "base_link" frame and safely plan around obstacles in its environment.

Suppose that we have the high level task described above of taking points in the "base_laser" frame

and transforming them to the "base_link" frame. The first thing we need to do is to create a node that will be responsible for publishing the transforms in our system. Next, we'll have to create a node to listen to the transform data published over ROS and apply it to transform a point. We'll start by creating a package for the source code to live in and we'll give it a simple name like "robot_setup_tf" We'll have dependencies on roscpp, tf, and geometry_msgs.

```
cd %TOP_DIR_YOUR_CATKIN_WS%/src

catkin_create_pkg robot_setup_tf roscpp tf geometry_msgs
```

Note that you have to run the command above where you have permission to do so (e.g. ~/ros where you might have created for the previous tutorials).

Alternative in fuerte, groovy and hydro: there is a standard robot_setup_tf_tutorial package in the navigation_tutorials stack. You may want to install by following (%YOUR_ROS_DISTRO% can be { fuerte, groovy } etc.):

```
sudo apt-get install ros-%YOUR_ROS_DISTRO%-navigation-tutorials
```

Broadcasting a Transform

Now that we've got our package, we need to create the node that will do the work of broadcasting the base_laser → base_link transform over ROS. In the robot_setup_tf package you just created, fire up your favorite editor and paste the following code into the src/tf_broadcaster.cpp file.

```
Toggle line numbers
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>

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

 ros::Rate r(100);

 tf::TransformBroadcaster broadcaster;

 while(n.ok()){
```

```
  broadcaster.sendTransform(
    tf::StampedTransform(
      tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.1, 0.0, 0.2)),
      ros::Time::now(),"base_link", "base_laser"));
  r.sleep();
 }
}
```

Now, let's take a look at the code that is relevant to publishing the base_link → base_laser transform in more detail.

```
#include <tf/transform_broadcaster.h>
```

The tf package provides an implementation of a tf::TransformBroadcaster to help make the task of publishing transforms easier. To use the TransformBroadcaster, we need to include the tf/transform_broadcaster.h header file.

Toggle line numbers

```
 tf::TransformBroadcaster broadcaster;
```

Here, we create a TransformBroadcaster object that we'll use later to send the base_link → base_laser transform over the wire.

Toggle line numbers

```
  broadcaster.sendTransform(
    tf::StampedTransform(
      tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.1, 0.0, 0.2)),
      ros::Time::now(),"base_link", "base_laser"));
```

This is where the real work is done. Sending a transform with a TransformBroadcaster requires five arguments. First, we pass in the rotation transform, which is specified by a btQuaternion for any rotation that needs to occur between the two coordinate frames. In this case, we want to apply no rotation, so we send in a btQuaternion constructed from pitch, roll, and yaw values equal to zero. Second, a btVector3 for any translation that we'd like to apply. We do, however, want to apply a translation, so we create a btVector3 corresponding to the laser's x offset of 10cm and z offset of 20cm from the robot base. Third, we need to give the transform being published a timestamp, we'll just stamp it with ros::Time::now(). Fourth, we need to pass the name of the parent node of the link we're creating, in this case "base_link." Fifth, we need to pass the name of the child node of the link we're creating, in this case "base_laser."

Using a Transform

Above, we created a node that publishes the base_laser → base_link transform over ROS. Now, we're going to write a node that will use that transform to take a point in the "base_laser" frame and transform it to a point in the "base_link" frame. Once again, we'll start by pasting the code below into a file and follow up with a more detailed explanation. In the robot_setup_tf package, create a file called src/tf_listener.cpp and paste the following:

Toggle line numbers

```
#include <ros/ros.h>
#include <geometry_msgs/PointStamped.h>
#include <tf/transform_listener.h>

void transformPoint(const tf::TransformListener& listener){
 //we'll create a point in the base_laser frame that we'd like to transform to the base_link frame
 geometry_msgs::PointStamped laser_point;
 laser_point.header.frame_id = "base_laser";

 //we'll just use the most recent transform available for our simple example
 laser_point.header.stamp = ros::Time();

 //just an arbitrary point in space
 laser_point.point.x = 1.0;
 laser_point.point.y = 0.2;
 laser_point.point.z = 0.0;

 try{
  geometry_msgs::PointStamped base_point;
  listener.transformPoint("base_link", laser_point, base_point);

  ROS_INFO("base_laser: (%.2f, %.2f. %.2f) -----> base_link: (%.2f, %.2f, %.2f) at time %.2f",
      laser_point.point.x, laser_point.point.y, laser_point.point.z,
      base_point.point.x, base_point.point.y, base_point.point.z, base_point.header.stamp.toSec());
 }
 catch(tf::TransformException& ex){
  ROS_ERROR("Received an exception trying to transform a point from \"base_laser\" to \"base_link
```

```
\": %s", ex.what());
  }
}

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

  tf::TransformListener listener(ros::Duration(10));

  //we'll transform a point once every second
  ros::Timer timer = n.createTimer(ros::Duration(1.0), boost::bind(&transformPoint, boost::ref(listene
r)));

  ros::spin();

}
```

```
#include <tf/transform_listener.h>
```

Here, include the tf/transform_listener.h header file that we'll need to create a tf::TransformListener. A TransformListener object automatically subscribes to the transform message topic over ROS and manages all transform data coming in over the wire.

```
void transformPoint(const tf::TransformListener& listener){
```

We'll create a function that, given a TransformListener, takes a point in the "base_laser" frame and transforms it to the "base_link" frame. This function will serve as a callback for the ros::Timer created in the main() of our program and will fire every second.

```
  //we'll create a point in the base_laser frame that we'd like to transform to the base_link frame
  geometry_msgs::PointStamped laser_point;
  laser_point.header.frame_id = "base_laser";

  //we'll just use the most recent transform available for our simple example
  laser_point.header.stamp = ros::Time();
```

```
//just an arbitrary point in space
laser_point.point.x = 1.0;
laser_point.point.y = 0.2;
laser_point.point.z = 0.0;
```

Here, we'll create our point as a geometry_msgs::PointStamped. The "Stamped" on the end of the message name just means that it includes a header, allowing us to associate both a timestamp and a frame_id with the message. We'll set the stamp field of the laser_point message to be ros::Time() which is a special time value that allows us to ask the TransformListener for the latest available transform. As for the frame_id field of the header, we'll set that to be "base_laser" because we're creating a point in the "base_laser" frame. Finally, we'll set some data for the point.... picking values of x: 1.0, y: 0.2, and z: 0.0.

```
try{
  geometry_msgs::PointStamped base_point;
  listener.transformPoint("base_link", laser_point, base_point);

  ROS_INFO("base_laser: (%.2f, %.2f. %.2f) -----> base_link: (%.2f, %.2f, %.2f) at time %.2f",
      laser_point.point.x, laser_point.point.y, laser_point.point.z,
      base_point.point.x, base_point.point.y, base_point.point.z, base_point.header.stamp.toSec());
}
```

Now that we have the point in the "base_laser" frame we want to transform it into the "base_link" frame. To do this, we'll use the TransformListener object, and call transformPoint() with three arguments: the name of the frame we want to transform the point to ("base_link" in our case), the point we're transforming, and storage for the transformed point. So, after the call to transformPoint(), base_point holds the same information as laser_point did before only now in the "base_link" frame.

```
catch(tf::TransformException& ex){
  ROS_ERROR("Received an exception trying to transform a point from \"base_laser\" to \"base_link
\": %s", ex.what());
}
```

If for some reason the base_laser → base_link transform is unavailable (perhaps the tf_broadcaster is not running), the TransformListener may throw an exception when we call transformPoint(). To make sure we handle this gracefully, we'll catch the exception and print out an error for the user.

Building the Code

Now that we've written our little example, we need to build it. Open up the CMakeLists.txt file that is autogenerated by roscreate-pkg and add the following lines to the bottom of the file.

```
add_executable(tf_broadcaster src/tf_broadcaster.cpp)

add_executable(tf_listener src/tf_listener.cpp)

target_link_libraries(tf_broadcaster ${catkin_LIBRARIES})

target_link_libraries(tf_listener ${catkin_LIBRARIES})
```

Next, make sure to save the file and build the package.

```
$ cd %TOP_DIR_YOUR_CATKIN_WS%

$ catkin_make
```

Running the Code

In the first terminal, run a core.

```
roscore
```

In the second terminal, we'll run our tf_broadcaster

```
rosrun robot_setup_tf tf_broadcaster
```

Now we'll use the third terminal to run our tf_listener to transform our mock point from the "base_laser" frame to the "base_link" frame.

```
rosrun robot_setup_tf tf_listener
```

If all goes well, you should see the following output showing a point being transformed from the "base_laser" frame to the "base_link" frame once a second.

[INFO] 1248138528.200823000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.20, 0.20) at time 1248138528.19

[INFO] 1248138529.200820000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.20, 0.20) at time 1248138529.19

[INFO] 1248138530.200821000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.20, 0.20) at time 1248138530.19

[INFO] 1248138531.200835000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.20, 0.20) at time 1248138531.19

[INFO] 1248138532.200849000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.20, 0.20) at time 1248138532.19

RESULT: