Tutorials Documentation

Lasi Piyathilaka

Apr 06, 2021

CONTENTS

CHAPTER

ONE

ENEX13004:WEEK 1 - SOFTWARE INSTALLATION

1.1 PC Setup

In this course we are utilizing a pre-configured virtual machine. The second option is to install a native Ubuntu machine with the required software. The virtual machine approach is by far the easiest option and ensures the fewest build errors during training but is limited in its ability to connect to certain hardware, particularly over USB.

1.1.1 Virtual Machine Configuration (Recommended)

The VM method is the most convenient method of utilizing the training materials: Follow the links below and install Virtual box software and ROS Melodic training VM

1. Download virtual box
2. Download ROS Melodic training VM
3. Import image into virtual box
4. Start virtual machine
    1. *Note: If possible, assign two cores in Settings>>System>>Processor to your virtual machine before starting your virtual machine. This setting can be adjusted when the virtual machine is closed and shut down.
5. Log into virtual machine, user: `ros-industrial`, pass: `rosindustrial` (no spaces or hyphens)

CHAPTER

TWO

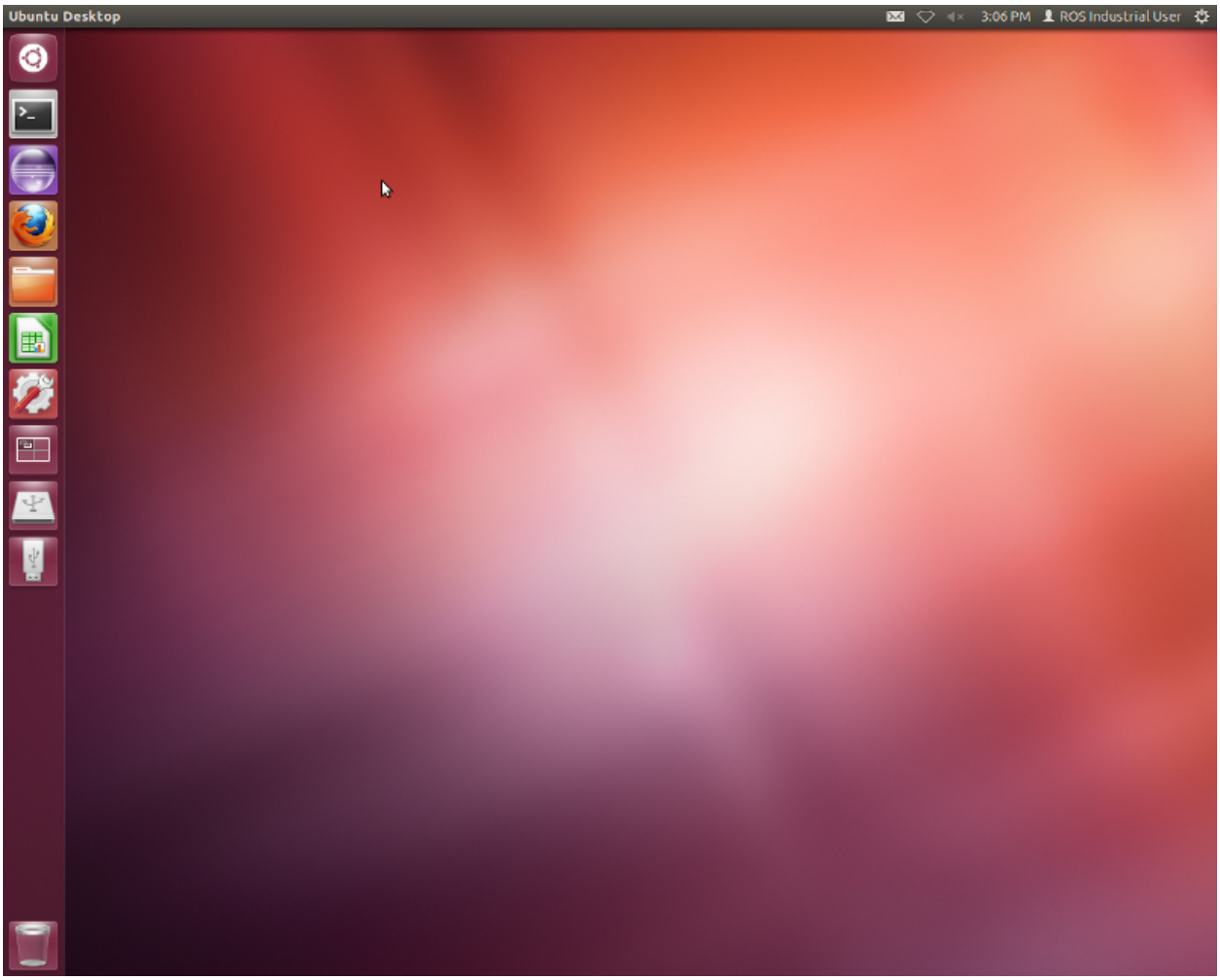
ENEX13004:WEEK 1 -UBUNTU GUI

2.1 Navigating the Ubuntu GUI

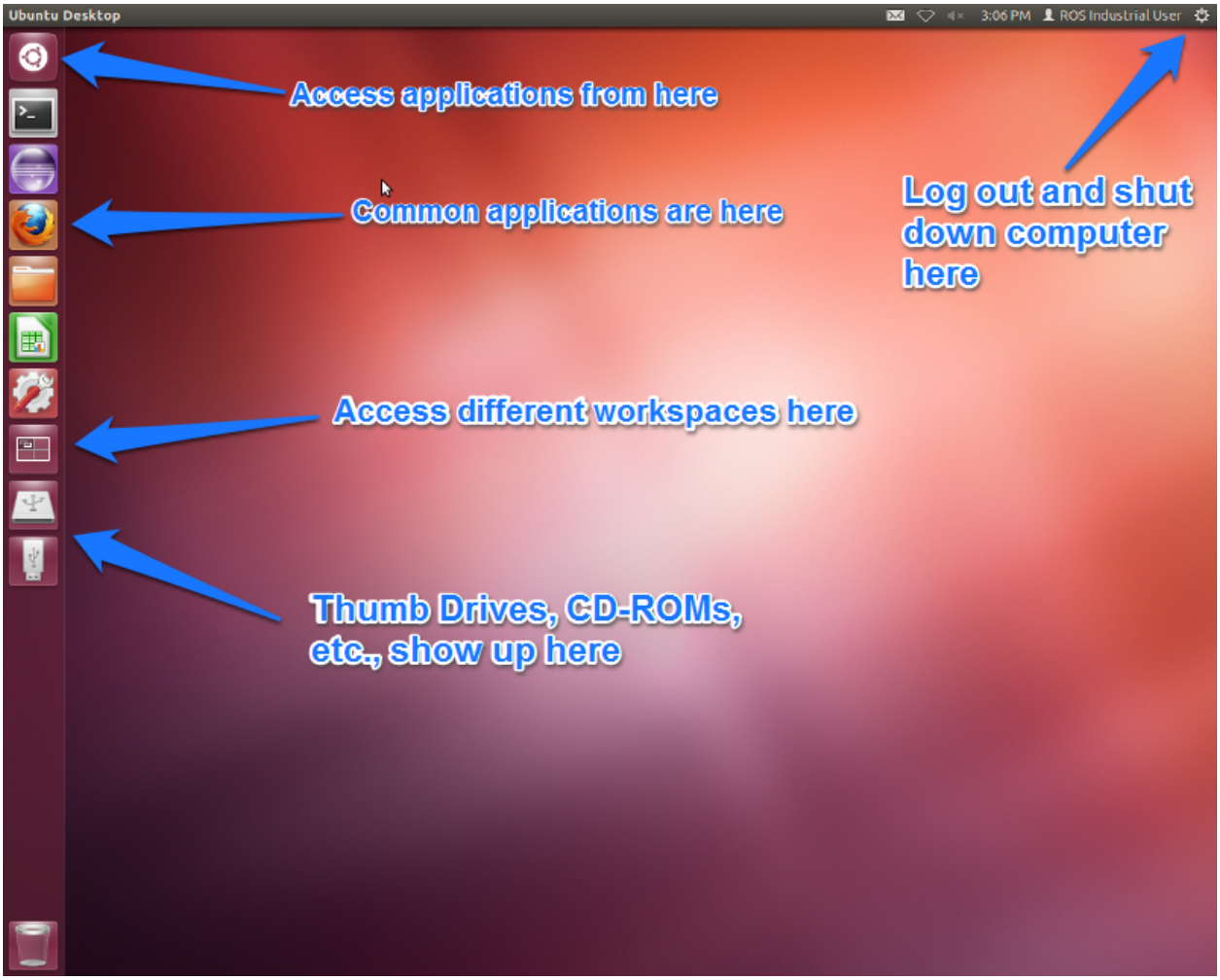
In this exercise, we will familiarize ourselves with the graphical user interface (GUI) of the Ubuntu operating system.

2.1.1 Task 1: Familiarize Yourself with the Ubuntu Desktop

At the log-in screen, click in the password input box, enter `rosindustrial` for the password, and hit enter. The screen should look like the image below when you log in:



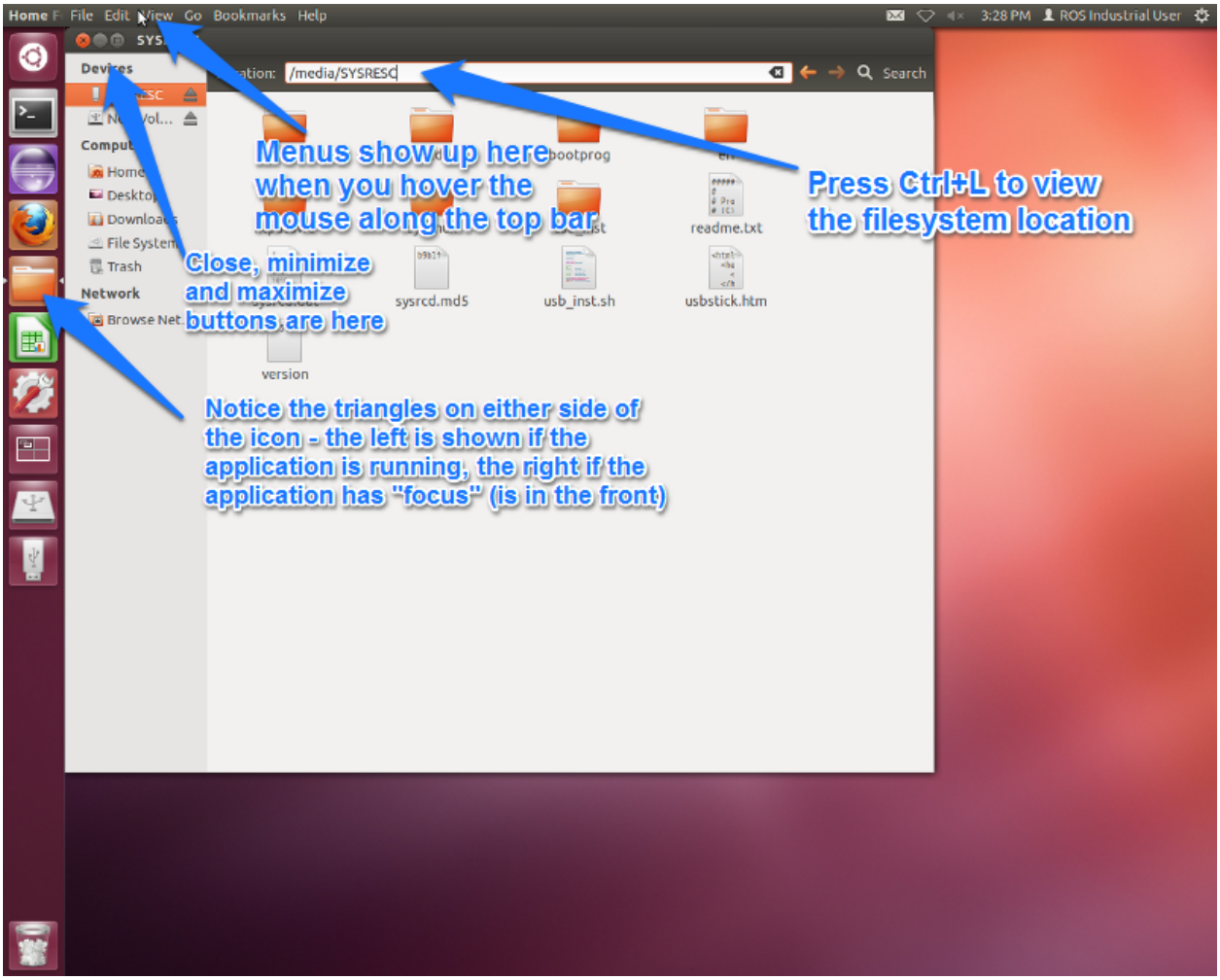
There are several things you will notice on the desktop:



1. The gear icon on the top right of the screen brings up a menu which allows the user to log out, shut down the computer, access system settings, etc…
2. The bar on the left side shows running and "favorite" applications, connected thumb drives, etc.
3. The top icon is used to access all applications and files. We will look at this in more detail later.
4. The next icons are either applications which are currently running or have been "pinned" (again, more on pinning later)
5. Any removable drives, like thumb drives, are found after the application icons.
6. If the launcher bar gets "too full", clicking and dragging up/down allows you to see the applications that are hidden.
7. To reorganize the icons on the launcher, click and hold the icon until it "pops out", then move it to the desired location.

2.1.2 Task 2: Open and Inspect an Application

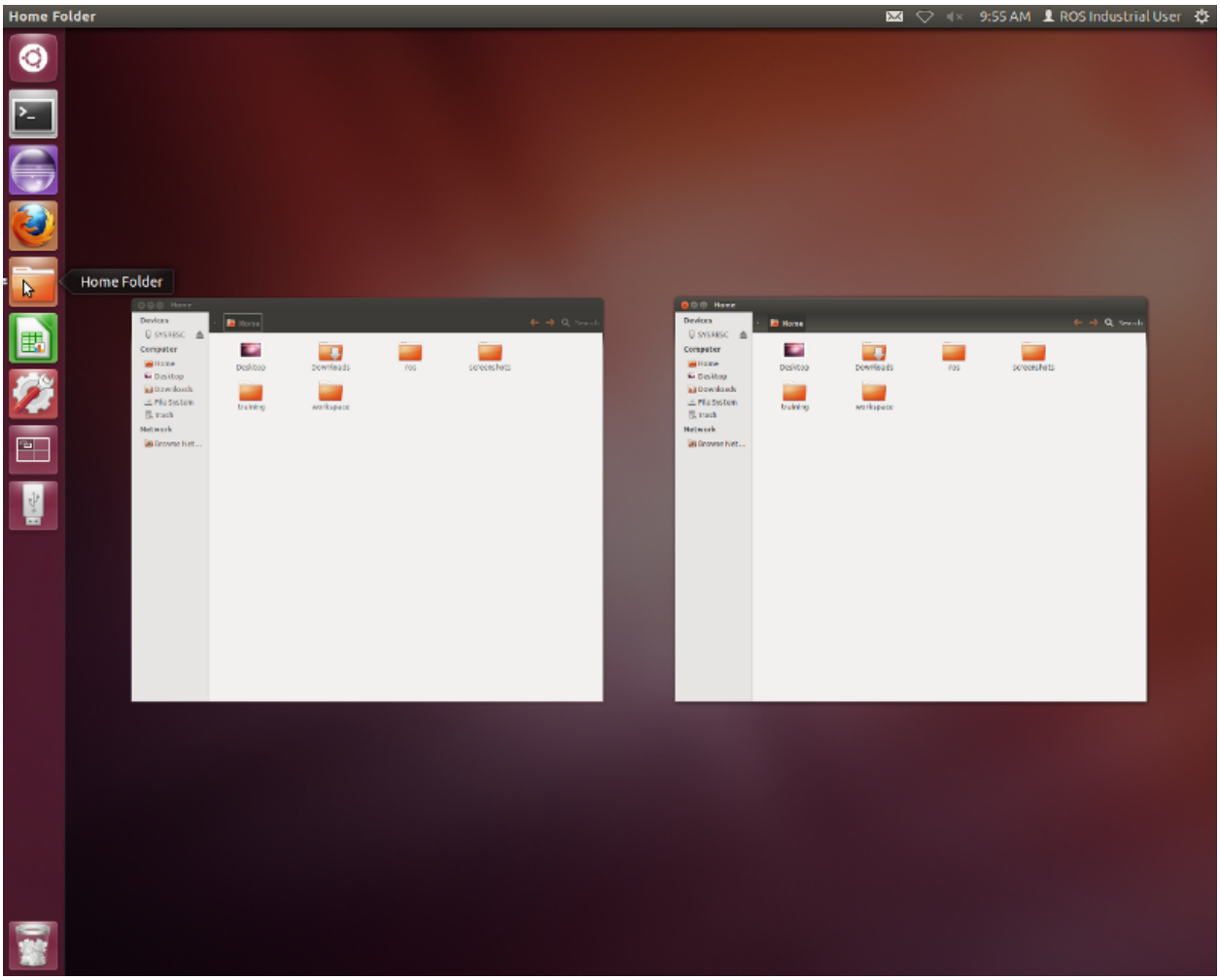
Click on the filing-cabinet icon in the launcher. A window should show up, and your desktop should look like something below:



Things to notice:

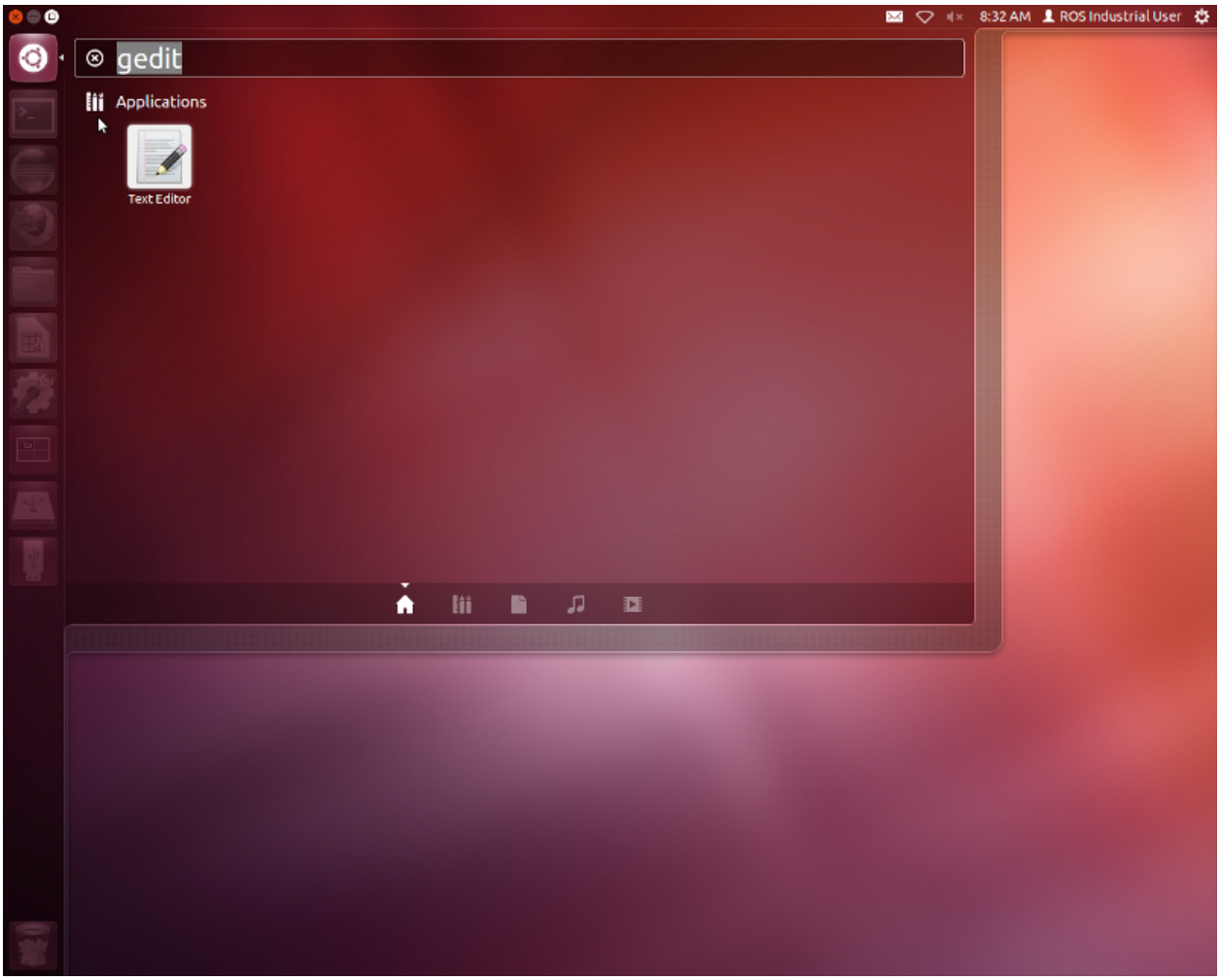
1. The close, minimize, and maximize buttons typically found on the right-hand side of the window title bar are found on the left-hand side.
2. The menu for windows are found on the menu bar at the top of the screen, much in the same way Macs do. The menus, however, only show up when you hover the mouse over the menu bar.
3. Notice that there are menu highlights of the folder icon. The dots on the left show how many windows of this application are open. Clicking on these icons when the applications are open does one of two things:

- If there is only one window open, this window gets focus.
- If more than one are open, clicking a second time causes all of the windows to show up in the foreground, so that you can choose which window to go to (see below):

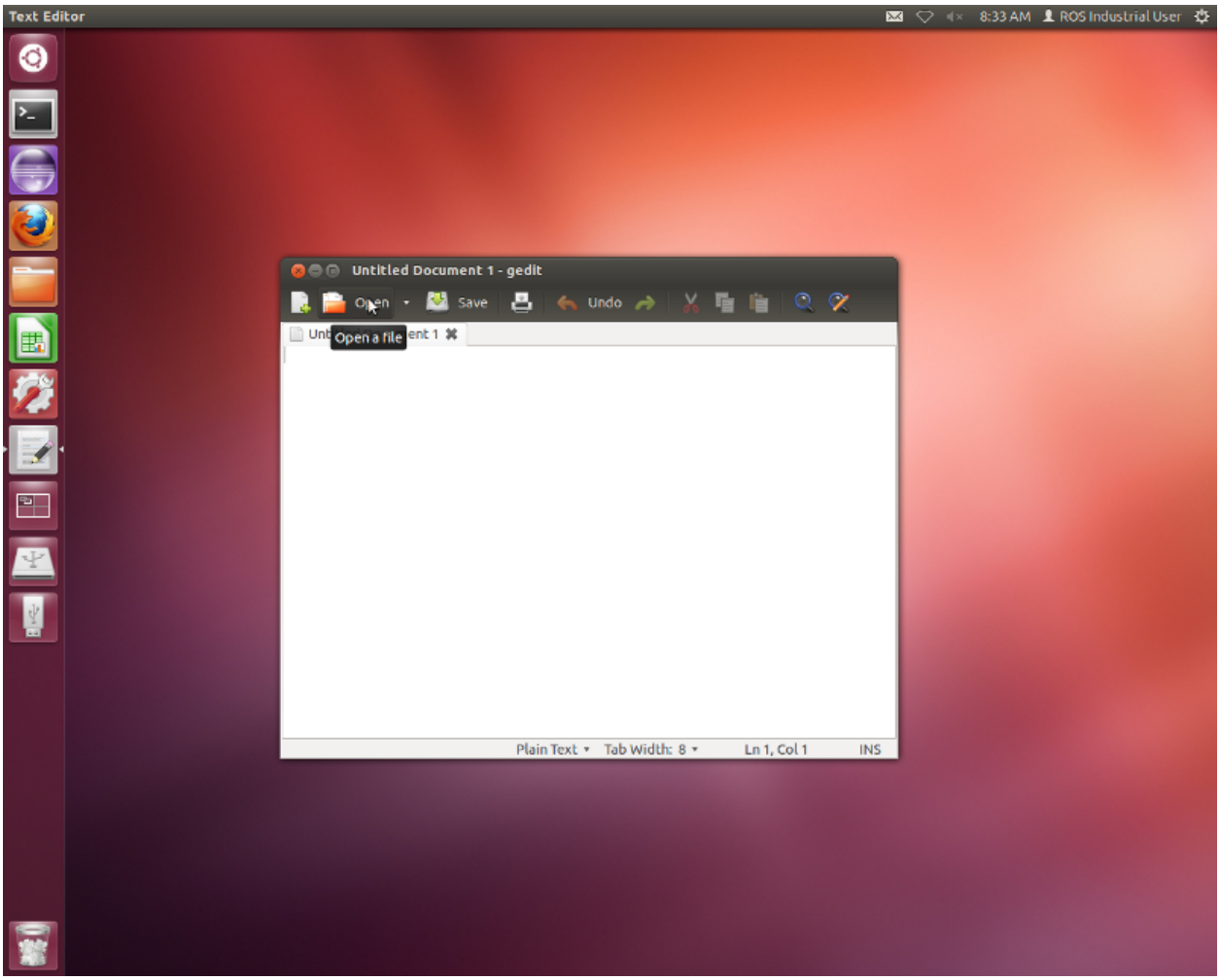


2.1.3 Task 3: Start an Application & Pin it to the Launcher Bar

Click on the launcher button (top left) and type gedit in the search box. The “Text Editor” application (this is actually gedit) should show up (see below):



Click on the application. The text editor window should show up on the screen, and the text editor icon should show up on the launcher bar on the left-hand side (see below):



1. Right-click on the text editor launch icon, and select “Lock to Launcher”.
2. Close the gedit window. The launcher icon should remain after the window closes.
3. Click on the gedit launcher icon. You should see a new gedit window appear.

CHAPTER

THREE

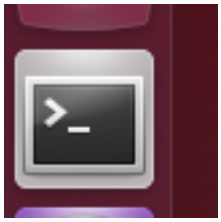
ENEX13004:WEEK 1- LINUX BASICS

3.1 The Linux Terminal

In this exercise, we will familiarize ourselves with the Linux terminal.

3.1.1 Starting the Terminal

1. To open the terminal, click on the terminal icon:



2. Create a second terminal window, either by:
   - Right-clicking on the terminal and selecting the "Open Terminal" or
   - Selecting "Open Terminal" from the "File" menu
3. Create a second terminal within the same window by pressing "Ctrl+Shift+T" while the terminal window is selected.
4. Close the 2nd terminal tab, either by:
   - clicking the small 'x' in the terminal tab (not the main terminal window)
   - typing `exit` and hitting enter.
5. The window will have a single line, which looks like this:

   ```
   ros-industrial@ros-i-melodic-vm:~$
   ```

6. This is called the prompt, where you enter commands. The prompt, by default, provides three pieces of information:
   1. *ros-industrial* is the login name of the user you are running as.
   2. *ros-i-melodic-vm* is the host name of the computer.
   3. ~ is the directory in which the terminal is currently in. (More on this later).
7. Close the terminal window by typing `exit` or clicking on the red 'x' in the window's titlebar.

3.1.2 Navigating Directories and Listing Files

Home Directory

The directory in which you find yourself when you first login is called your home directory.You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.

1. You can go in your home directory anytime using the following command ~.

```
*$cd ~
```

1. Here ~ indicates the home directory. Suppose you have to go in any other user's home directory, use the following command

```
$cd ~username
```

1. To go in your last directory, you can use the following command

```
$cd ..
```

Absolute/Relative Pathnames

Directories are arranged in a hierarchy with root (/) at the top. The position of any file within the hierarchy is described by its pathname.

Elements of a pathname are separated by a /. A pathname is absolute, if it is described in relation to root, thus absolute pathnames always begin with a /.

Following are some examples of absolute filenames.

```
/etc/passwd /dev/rdsk/Os3
```

A pathname can also be relative to your current working directory. Relative pathnames never begin with /. Relative to user's home directory, some pathnames might look like this

```
chem/notes personal/res
```

ls Command

1. Go to the home directory `$cd ~` and enter `ls` into the terminal.
   - You should see files like `test.txt`.
   - Directories, like `Desktop`, are colored in blue.
   - The file `sample_job` is in green; this indicates it has its "execute" bit set, which means it can be executed as a command.
2. Type `ls *.txt`. Only the file `test.txt` will be displayed. This will display all the files with .txt extension
3. Enter `ls -l` into the terminal.
   - Adding the `-l` option shows one entry per line, with additional information about each entry in the directory.
   - The first 10 characters indicate the file type and permissions
   - The first character is `d` if the entry is a directory.
   - The next 9 characters are the permissions bits for the file
   - The third and fourth fields are the owning user and group, respectively.

- The second-to-last field is the time the file was last modified.
- If the file is a symbolic link, the link's target file is listed after the link's file name.

4. Enter `ls -a` in the terminal.
   - You will now see additional files, that starts with ".". These are hidden files
5. Enter `ls -a -l` (or `ls -al`) in the command.
   - You'll now see all details of each file such as the creation date and etc.

`pwd` and `cd` Commands

1. Enter `pwd` into the terminal.
   - This will show you the full path of the directory you are working in.To determine where you are within the filesystem hierarchy at any time, enter the command pwd to print the current working directory
2. Enter `cd new` into the terminal.
   - The prompt should change to `ros-industrial@ros-i-melodic-vm:~/new$`.
   - Typing `pwd` will show you the path to the current directory the directory `/home/ros-industrial/new`.
3. Enter `cd ..` into the terminal to go back to the previous directory
4. Enter `cd /bin`, followed by `ls`.
   - This folder contains a list of the most basic Linux commands. *Note that `pwd` and `ls` are both in this folder.*
5. Enter `cd ~/new` to return to our working directory.
   - Linux uses the ~ character as a shorthand representation for your home directory.
   - It's a convenient way to reference files and paths in command-line commands.
   - You'll be typing it a lot in this class... remember it!

If you want a full list of options available for any of the commands given in this section, type `man <command>` (where `<command>` is the command you want information on) in the command line. This will provide you with built-in documentation for the command. Use the arrow and page up/down keys to scroll, and `q` to exit.

3.1.3 Altering Files

mv Command

1. Type `mv test.txt test2.txt`, followed by `ls`.
   - You will notice that the file has been renamed to `test2.txt`. *This step shows how `mv` can rename files.*
2. Type `mv test2.txt new`, then `ls`.
   - The file will no longer be present in the folder.
3. Type `cd new`, then `ls`.
   - You will see `test2.txt` in the folder. *These steps show how `mv` can move files.*
4. Type `mv test2.txt ../test.txt`, then `ls`.
   - `test2.txt` will no longer be there.
5. Type `cd ..`, then `ls`.

- You will notice that `test.txt` is present again. *This shows how `mv` can move and rename files in one step.*

cp Command

1. Type `cp test.txt new/test2.txt`, then `ls new`.
   - You will see `test2.txt` is now in the `new` folder.
2. Type `cp test.txt "test copy.txt"`, then `ls -l`.
   - You will see that `test.txt` has been copied to `test copy.txt`. *Note that the quotation marks are necessary when spaces or other special characters are included in the file name.*

rm Command

1. Type `rm "test copy.txt"`, then `ls -l`.
   - You will notice that `test copy.txt` is no longer there.

mkdir Command

1. Type `mkdir new2`, then `ls`.
   - You will see there is a new folder `new2`.

touch Command

1. Type `touch ~/Templates/"Untitled Document"`.
   - This will create a new Document named **"Untitled Document"**

You can use the `-i` flag with `cp`, `mv`, and `rm` commands to prompt you when a file will be overwritten or removed.

3.1.4 Job management

Editing Text (and Other GUI Commands)

1. Type `gedit test.txt`.
   - You will notice that a new text editor window will open, and `test.txt` will be loaded.
   - The terminal will not come back with a prompt until the window is closed.
2. There are two ways around this limitation. Try both...
3. **Starting the program and immediately returning a prompt:**
   1. Type `gedit test.txt &`.
      - The `&` character tells the terminal to run this command in "the background", meaning the prompt will return immediately.
   2. Close the window, then type `ls`.
      - In addition to showing the files, the terminal will notify you that `gedit` has finished.
4. **Moving an already running program into the background:**

1. Type `gedit test.txt`.
   - The window should open, and the terminal should not have a prompt waiting.
2. In the terminal window, press Ctrl+Z.
   - The terminal will indicate that `gedit` has stopped, and a prompt will appear.
3. Try to use the `gedit` window.
   - Because it is paused, the window will not run.
4. Type `bg` in the terminal.
   - The `gedit` window can now run.
5. Close the `gedit` window, and type `ls` in the terminal window.
   - As before, the terminal window will indicate that `gedit` is finished.

Running Commands as Root

1. In a terminal, type `ls -a /root`.
   - The terminal will indicate that you cannot read the folder `/root`.
   - Many times you will need to run a command that cannot be done as an ordinary user, and must be done as the "super user"
2. To run the previous command as root, add `sudo` to the beginning of the command.
   - In this instance, type `sudo ls -a /root` instead.
   - The terminal will request your password (in this case, `rosindustrial`) in order to proceed.
   - Once you enter the password, you should see the contents of the `/root` directory.

***Warning**: `sudo` is a powerful tool which doesn't provide any sanity checks on what you ask it to do, so be **VERY** careful in using it.*

CHAPTER

FOUR

ENEX13004:WEEK 1- ROS NODE

```
Note: This tutorial assumes that you have completed the previous tutorials: building a
ROS package (/ROS/Tutorials/BuildingPackages).
```

```
Please ask about problems and questions regarding this tutorial on answers.ros.org
(http://answers.ros.org). Don't forget to include in your question the link to this␣
↪page, the
versions of your OS & ROS, and also add appropriate tags.
```

4.1 Understanding ROS Nodes

Description: This tutorial introduces ROS graph concepts and discusses the use of roscore (/roscore), rosnode (/rosnode), and rosrun (/rosrun) commandline tools.

Tutorial Level: BEGINNER

Next Tutorial: Understanding ROS topics (/ROS/Tutorials/UnderstandingTopics)

```
Contents
1. Prerequisites
2. Quick Overview of Graph Concepts
3. Nodes
4. Client Libraries
5. roscore
6. Using rosnode
7. Using rosrun
8. Review
```

4.1.1 1. Prerequisites

For this tutorial we'll use a lighweight simulator, to install it run the following command:

```
$ sudo apt-get install ros-<distro>-ros-tutorials
```

Replace '' with the name of your ROS distribution (e.g. indigo, jade, kinetic)

4.1.2 2. Quick Overview of Graph Concepts

```
Nodes (/Nodes): A node is an executable that uses ROS to communicate with other
nodes.
Messages (/Messages): ROS data type used when subscribing or publishing to a topic.
Topics (/Topics): Nodes can publish messages to a topic as well as subscribe to a␣
↪topic to
receive messages.
```

```
Master (/Master): Name service for ROS (i.e. helps nodes find each other)
rosout (/rosout): ROS equivalent of stdout/stderr
roscore (/roscore): Master + rosout + parameter server (parameter server will be
introduced later)
```

4.1.3 3. Nodes

A node really isn't much more than 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.

4.1.4 4. Client Libraries

ROS client libraries allow nodes written in different programming languages to communicate:

```
rospy = python client library
roscpp = c++ client library
```

4.1.5 5. roscore

roscore is the first thing you should run when using ROS.

Please run:

```
$ roscore
```

You will see something similar to:

```
... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/roslaunch-mac
hine_name-13039.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

```
started roslaunch server http://machine_name:33919/
ros_comm version 1.4.
```

```
SUMMARY
======
```

```
PARAMETERS
* /rosversion
* /rosdistro
```

```
NODES
```

```
auto-starting new master
process[master]: started with pid [13054]
ROS_MASTER_URI=http://machine_name:11311/
```

```
setting /run_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
process[rosout-1]: started with pid [13067]
started core service [/rosout]
```

If roscore does not initialize, you probably have a network configuration issue. See Network Setup - Single Machine Configuration (http://www.ros.org/wiki/ROS/NetworkSetup#Single_machine_configuration)

If roscore does not initialize and sends a message about lack of permissions, probably the ~/.ros folder is owned by root, change recursively the ownership of that folder with:

```
$ sudo chown -R <your_username> ~/.ros
```

4.1.6 6. Using rosnode

Open up a **new terminal** , and let's use **rosnode** to see what running roscore did... Bear in mind to keep the previous terminal open either by opening a new tab or simply minimizing it.

```
Note: When opening a new terminal your environment is reset and your ~/.bashrc file is
sourced. If you have trouble running commands like rosnode then you might need to add
some environment setup files to your ~/.bashrc or manually re-source them.
```

rosnode displays information about the ROS nodes that are currently running. The rosnode list command lists these active nodes:

```
$ rosnode list
```

```
You will see:
```

```
/rosout
```

This showed us that there is only one node running: rosout (/rosout). This is always running as it collects and logs nodes' debugging output.

The rosnode info command returns information about a specific node.

```
$ rosnode info /rosout
```

This gave us some more information about rosout, such as the fact that it publishes /rosout_agg.

```
--------------------------------------------------------------------------------
Node [/rosout]
Publications:
* /rosout_agg [rosgraph_msgs/Log]
```

```
Subscriptions:
* /rosout [unknown type]
```

```
Services:
* /rosout/get_loggers
* /rosout/set_logger_level
```

```
contacting node http://machine_name:54614/ ...
Pid: 5092
```

Now, let's see some more nodes. For this, we're going to use rosrun to bring up another node.

4.1.7 7. Using rosrun

rosrun allows you to use the package name to directly run a node within a package (without having to know the package path).

Usage:

```
$ rosrun [package_name] [node_name]
```

So now we can run the turtlesim_node in the turtlesim package.

Then, in a **new terminal** :

```
$ rosrun turtlesim turtlesim_node
```

You will see the turtlesim window:

NOTE: The turtle may look different in your turtlesim window. Don't worry about it - there are many types of turtle (/Distributions#Current_Distribution_Releases) and yours is a surprise!

In a **new terminal** :

```
$ rosnode list
```

You will see something similar to:

```
/rosout
/turtlesim
```

One powerful feature of ROS is that you can reassign Names from the command-line.

Close the turtlesim window to stop the node (or go back to the rosrun turtlesim terminal and use ctrl-C). Now let's re-run it, but this time use a Remapping Argument (/Remapping%20Arguments) to change the node's name:

```
$ rosrun turtlesim turtlesim_node __name:=my_turtle
```

Now, if we go back and use rosnode list:

```
$ rosnode list
```

```
You will see something similar to:
```

```
/my_turtle
/rosout
```

```
Note: If you still see /turtlesim in the list, it might mean that you stopped the␣
↪node in the
terminal using ctrl-C instead of closing the window, or that you don't have the
$ROS_HOSTNAME environment variable defined as described in Network Setup - Single
Machine Configuration
(http://www.ros.org/wiki/ROS/NetworkSetup#Single_machine_configuration). You can try
cleaning the rosnode list with: $ rosnode cleanup
```

We see our new /my_turtle node. Let's use another rosnode command, ping, to test that it's up:

```
$ rosnode ping my_turtle
```

```
rosnode: node is [/my_turtle]
pinging /my_turtle with a timeout of 3.0s
xmlrpc reply from http://aqy:42235/ time=1.152992ms
xmlrpc reply from http://aqy:42235/ time=1.120090ms
xmlrpc reply from http://aqy:42235/ time=1.700878ms
xmlrpc reply from http://aqy:42235/ time=1.127958ms
```

4.1.8 8. Review

What was covered:

```
roscore = ros+core : master (provides name service for ROS) + rosout (stdout/stderr) +
parameter server (parameter server will be introduced later)
rosnode = ros+node : ROS tool to get information about a node.
rosrun = ros+run : runs a node from a given package.
```

Now that you understand how ROS nodes work, let's look at how ROS topics work (/ROS/Tutorials/UnderstandingTopics). Also, feel free to press Ctrl-C to stop turtlesim_node.

```
Wiki: ROS/Tutorials/UnderstandingNodes (last edited 2019-06-05 23:41:12 by LukeMeier␣
↪(/LukeMeier))
```

CHAPTER

FIVE

ENEX13004:WEEK 2- PYTHON TURTLESIM EXAMPLE

See the following python script.

```
#!/usr/bin/env python
import math
import rospy
import turtlesim.srv
from geometry_msgs.msg import Twist

rospy.init_node('robot_letter', anonymous=True)
velocity_publisher = rospy.Publisher('/turtle1/cmd_vel', Twist, queue_size=10)
vel_msg = Twist()
speed = 1
angular_velocity= 1


# Initial velocity values for the publisher
vel_msg.linear.x = 0
vel_msg.linear.y = 0
vel_msg.linear.z = 0
vel_msg.angular.x = 0
vel_msg.angular.y = 0
vel_msg.angular.z = 0

def move():



    #Setting the current time for distance calculus
    t0 = rospy.Time.now().to_sec()
    move_straight(2,t0)


    t0 = rospy.Time.now().to_sec()
    rotate_angle(math.pi/2, t0)

    t0 = rospy.Time.now().to_sec()
    move_straight(3,t0)


    t0 = rospy.Time.now().to_sec()
    move_circle(math.pi,t0)
```

(continues on next page)

(continued from previous page)

```
    #create a new robot
    create_second_robot()


# a function to move the robot forward
def move_straight(move_distance,start_time):
    vel_msg.linear.x = speed
    vel_msg.angular.z = 0
    current_distance = 0
    #Loop to move the turtle in an specified distance
    while(current_distance < move_distance):
            #Publish the velocity
            velocity_publisher.publish(vel_msg)
            #Takes actual time to velocity calculus
            current_time=rospy.Time.now().to_sec()
            #Calculates distancePoseStamped
            current_distance= speed*(current_time-start_time)
            print("curent distance %d",current_distance)
    stop_robot()

#a function to roate the robot  angle
def rotate_angle(rotate_angle,start_time):
    vel_msg.linear.x = 0
    vel_msg.angular.z = angular_velocity
    current_angle = 0
    #Loop to move the turtle in an specified angle
    while(current_angle < rotate_angle):
            #Publish the velocity
            velocity_publisher.publish(vel_msg)
            #Takes actual time to velocity calculus
            current_time=rospy.Time.now().to_sec()
            #Calculates angle PoseStamped
            current_angle= angular_velocity*(current_time-start_time)
            print("curent distance %d",current_angle)
    stop_robot()

# force the robot to stop
def stop_robot():
    vel_msg.linear.x = 0
    vel_msg.angular.z = 0
    velocity_publisher.publish(vel_msg)

#move the robot in a curvey path
def move_circle(rotate_angle,start_time):
    vel_msg.linear.x = 1
    vel_msg.angular.z = angular_velocity
    current_angle = 0
    #Loop to move the turtle in an specified angle
    while(current_angle < rotate_angle):
            #Publish the velocity
            velocity_publisher.publish(vel_msg)
            #Takes actual time to velocity calculus
            current_time=rospy.Time.now().to_sec()
            #Calculates angle PoseStamped
            current_angle= angular_velocity*(current_time-start_time)
            print("curent distance %d",current_angle)
    stop_robot()
```

(continues on next page)

(continued from previous page)

```
#create the second turtlebot
def create_second_robot():
    rospy.wait_for_service('spawn')
    spawner = rospy.ServiceProxy('spawn', turtlesim.srv.Spawn)
    spawner(4, 2, 0, 'turtle2')

if __name__ == '__main__':
    try:
        #Testing our function
        move()
    except rospy.ROSInterruptException: pass
```

CHAPTER

SIX

ENEX13004:WEEK 2- ROS WORKSPACE SETUP

6.1 Create Catkin Workspace

In this exercise, we will create a ROS catkin workspace.

6.1.1 Create a Catkin Workspace

1. Create the root workspace directory (we'll use `catkin_ws`)

```
cd ~/
mkdir --parents catkin_ws/src
cd catkin_ws
```

2. Initialize the catkin workspace

```
catkin init
```

   - *Look for the statement "Workspace configuration appears valid", showing that your catkin workspace was created successfully. If you forgot to create the `src` directory, or did not run `catkin init` from the workspace root (both common mistakes), you'll get an error message like "WARNING: Source space does not yet exist".*

3. Build the workspace. This command may be issued anywhere under the workspace root-directory (i.e. `catkin_ws`).

```
catkin make
ls
```

   - *See that the `catkin_ws` directory now contains additional directories (build, devel, logs).*

4. These new directories can be safely deleted at any time (either manually, or using `catkin clean`). Note that catkin never changes any files in the `src` directory. Re-run `catkin build` to re-create the build/devel/logs directories.

```
catkin clean
ls
catkin make
ls
```

5. Make the workspace visible to ROS. Source the setup file in the devel directory.

```
source devel/setup.bash
```

- *This file MUST be sourced for every new terminal.*
- To save typing, add this to your `~/.bashrc` file, so it is automatically sourced for each new terminal:
  1. `gedit ~/.bashrc`
  2. add to the end: `source ~/catkin_ws/devel/setup.bash`
  3. save and close the editor

serial_manipulator.md

CHAPTER

SEVEN

ENEX13004:WEEK 3- ROBOTIC ARM SIMULATION WITH RVIZ AND URDF

7.1 Creating a 3 Link Robotic Arm Using URDF

In this exercise, we will build a 3 Link Robotic Manipulator using the Unified Robotic Description Format (URDF). It is an XML file format used in ROS to describe all elements of a robot and can be used to define a robot to its component level. We will then use RVIZ to simulate that robot.

First Let's create a ros package to define the URDF definitions.

1. Goto your `catkin_ws` and to the `src` folder and create a package

```
catkin_create_pkg serial_link_robot tf2 tf2_ros roscpp rospy turtlesim
```

   then do `catkin_make` to build the project. Now you have project folder to create URDF

2. Goto the project folder and build a urdf folder to include the urdf files

```
cd ~/catkin_ws/src/serial_link_robot
mkdir urdf
```

3. Now lets create our first URDF file. Inside the urdf folder create a file called'serial_link_robot.urdf'

```
gedit serial_link_robot.urdf
```

4. Now let's create our first link of the robot. Copy the following xml commands to the 'serial_link_robot.urdf' file..

```
 <?xml version='1.0'?>
<robot name="serial_Link_robot">
  <!-- Base Link -->
  <link name="base_link">
    <visual>
      <origin xyz="0 0 0.1" rpy="0 0 0" />
      <geometry>
      <box size="0.4 0.4 0.2"/>
      </geometry>
      <material name="gray">
      <color rgba="0.5 0.5 0.5 0.5"/>
      </material>
    </visual>
  </link>
</robot>
```

This will create the base link with the dimension 40 cm x 40 cm x 20 cm.

More information about URDF link elemenr can be found here http://wiki.ros.org/urdf/XML/link. Try to understand what each of these xml components means.

5. Now let's create a launch file to display the link in RVIZ. It is a 3D visulaisation software. Fist create a launch folder inside the package folder

```
cd ~/catkin_ws/src/serial_link_robot
mkdir launch
cd launch
```

6. Now let's create the launch file. Inside the `launch` folder create a file called ' serial_link_robot_rviz.launch'.

```
gedit serial_link_robot_rviz.launch
```

7. Now add the following launch commands to the `serial_link_robot_rviz.launch` file.

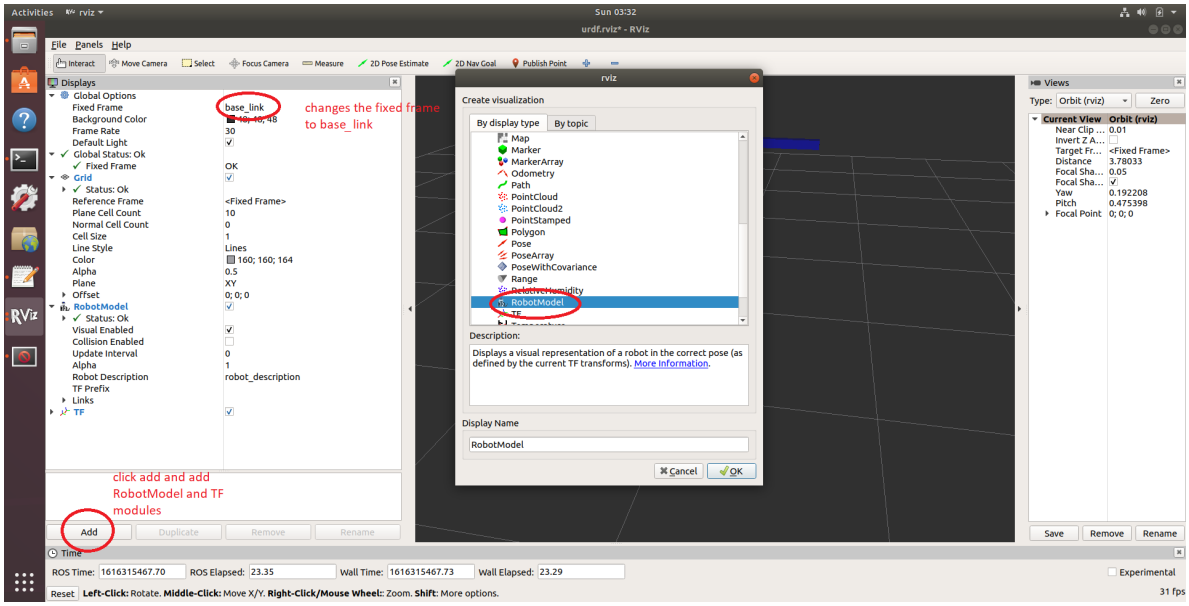
```
<launch>
  <!-- values passed by command line input -->
  <arg name="model" />
  <arg name="gui" default="False" />

<!-- set these parameters on Parameter Server -->
<param name="robot_description"
textfile="$(find serial_link_robot)/urdf/$(arg model)"
/>

<param name="use_gui" value="$(arg gui)"/>
<!-- Start 3 nodes: joint_state_publisher,
  robot_state_publisher and rviz -->

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

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


<node name="rviz" pkg="rviz" type="rviz"
  args="-d $(find serial_link_robot)/urdf.rviz"
  required="true" />
</launch>
```

8. This roslaunch file performs the following:
   - Loads the model specified in the command line into the Parameter Server.
   - Starts nodes that publish the JointState and transforms.
   - Starts rviz with a configuration file (urdf.rviz).
9. Now you can run the launch file by runing the roslaunch command in the terminal

   roslaunch serial_link_robot serial_link_robot_rviz.launch model:=serial_link_robot.urdf

10. This will open the Rviz window. Click on 'ADD' button (left bottom) and add 'RobotModel' and 'TF' modules. Also, change the 'Fixed Frame' in Global Option from map to base_link. This is the global reference frame.

Save the rviz file before closing the window.



11. Now let's add a joint and another link. More information about URDF joint element can be found here http://wiki.ros.org/urdf/XML/joint. Its a revolute joint that rotates from -pi to +pi. See how I have added the color to the link. Also, see how I have assigned the origin of the visual component of the link. This is equal to half of the link length. Open the urdf file and add the followings before the `</robot>` tag.

```
    <!-- Joint 1 -->
  <joint name="joint_1" type="revolute">
    <parent link="base_link"/>
    <child link="link_1"/>
    <origin xyz="0 0 0.2" rpy="0 0 0" />
    <axis xyz="0 0 1" />
    <limit effort="100.0" velocity="0.5" lower="-3.14"
  upper="3.14"/>
  </joint>

<!-- Link 1 -->
  <link name="link_1">
    <visual>
      <origin xyz="0 0 0.2" rpy="0 0 0" />
      <geometry>
      <box size="0.05 0.05 0.4"/>
      </geometry>
      <material name="yellow">
       <color rgba="1.0 1.0 0 0.5"/>
      </material>
    </visual>
  </link>
```

12. Now let's display and move the robot. Run the following command in a terminal'

    roslaunch serial_link_robot serial_link_robot_rviz.launch model:=serial_link_robot.urdf gui:=True

    The `gui` parameter open a joint state publisher window. You can manupulate the robot by chaning the joint slider bars.

13. Now lets expand the robot to a 3 link manupulator. Let's add more joints links. see the last link configuration.

Its in y direction initially

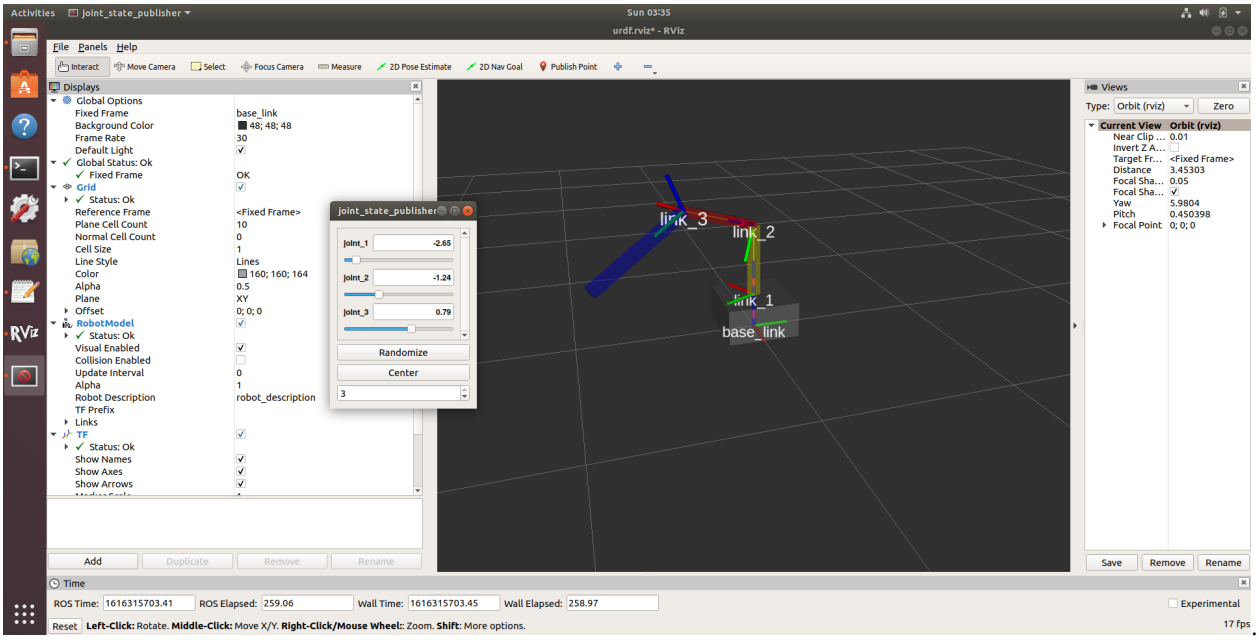
See the recoded zoom class for the demo. The completed urdf file is shown below.

```
<?xml version='1.0'?>
<robot name="serial_Link_robot">
  <!-- Base Link -->
  <link name="base_link">
    <visual>
      <origin xyz="0 0 0.1" rpy="0 0 0" />
      <geometry>
      <box size="0.4 0.4 0.2"/>
      </geometry>
      <material name="gray">
      <color rgba="0.5 0.5 0.5 0.5"/>
      </material>
    </visual>
  </link>

<!-- Joint 1 -->
  <joint name="joint_1" type="revolute">
    <parent link="base_link"/>
    <child link="link_1"/>
    <origin xyz="0 0 0.2" rpy="0 0 0" />
    <axis xyz="0 0 1" />
    <limit effort="100.0" velocity="0.5" lower="-3.14"
  upper="3.14"/>
  </joint>

<!-- Link 1 -->
  <link name="link_1">
    <visual>
      <origin xyz="0 0 0.2" rpy="0 0 0" />
      <geometry>
      <box size="0.05 0.05 0.4"/>
      </geometry>
      <material name="yellow">
       <color rgba="1.0 1.0 0 0.5"/>
      </material>
    </visual>
  </link>

<!-- Joint 2 -->
  <joint name="joint_2" type="revolute">
    <parent link="link_1"/>
    <child link="link_2"/>
    <origin xyz="0 0 0.4" rpy="0 0 0" />
    <axis xyz="1 0 0" />
    <limit effort="100.0" velocity="0.5" lower="-3.14"
  upper="3.14"/>
  </joint>

<!-- Link 2 -->
<link name="link_2">
    <visual>
      <origin xyz="0 0 0.25" rpy="0 0 0" />
      <geometry>
      <box size="0.05 0.05 0.5"/>
```

(continues on next page)

(continued from previous page)

```
        </geometry>
        <material name="Red">
         <color rgba="1 0 0 0.5"/>
        </material>
      </visual>
    </link>

  <!-- Joint 3 -->
   <joint name="joint_3" type="revolute">
      <parent link="link_2"/>
      <child link="link_3"/>
      <origin xyz="0 0 0.5" rpy="0 0 0" />
      <axis xyz="1 0 0" />
      <limit effort="100.0" velocity="0.5" lower="-3.14"
    upper="3.14"/>
    </joint>

  <!-- Link 3 -->
  <link name="link_3">
      <visual>
        <origin xyz="0 0.3 0" rpy="0 0 0" />
        <geometry>
        <box size="0.05 0.6 0.05"/>
        </geometry>
        <material name="Blue">
         <color rgba="0 0 1 0.5"/>
        </material>
      </visual>
    </link>

  </robot>
```



1. You can launch the launch file using the command shown above and manipulate the your Robot arm.

CHAPTER

EIGHT

ENEX13004:WEEK 3- ROBOTIC ARM SIMULATION WITH CODES

8.1 Simulating the Robotic Arm Using software codes in RVIZ

In this exercise, we are going to write a python script to simulate the Robotic arm that we built in the last tutorial. Previously, we used ROS inbuilt joint_state_publisher (http://wiki.ros.org/joint_state_publisher) package with GUI joint controller to simulate the robot. Now let's do the same with our own package.

First Let's create a ros package to define the URDF definitions.

1. Read the `robot_state_publisher` documentation (http://wiki.ros.org/robot_state_publisher). This is the main package that read the URDF definitions from the URDF file and converts it to `tf` messages. To get the joint positions (joints angles), it is subscribing to the `joint_state` topic. It has the message type of `JointState` (http://docs.ros.org/en/api/sensor_msgs/html/msg/JointState.html). Read the documentation for the `JointState` message type. We need to publish the joint angles as an array.

2. Let's write a python script to send joint angles to our robot. Creat a `joint_publisher.py` script inside the `src` folder of the `serial_link_robot` package.See how I have assigned each component of the joit_state_msg. The joint name should be the same as how you defined each link in the URDF file. The `joint_state_msg.position` variable assigns the angle values in radians.

```
#!/usr/bin/env python
import rospy

from sensor_msgs.msg import JointState
from std_msgs.msg import Header


def publish_joint_angles():

    rospy.init_node('robot_joint_publisher', anonymous=True)
    joint_publisher =rospy.Publisher('/joint_states',JointState,queue_size=10)
    joint_state_msg = JointState()

    # this is the rate of publishing. Need to have rate.sleep() command inside␣
↪the while loop
    rate = rospy.Rate(10) # 10hz

    joint_state_msg.header = Header()
    joint_state_msg.name = ['joint_1', 'joint_2', 'joint_3']
    # angles are in radians
    joint_state_msg.position = [0.5, 0.5, 1.2]
    joint_state_msg.velocity = []
    joint_state_msg.effort = []
```

(continues on next page)

(continued from previous page)

```
    # this while loop will continuously publish the joint states
    while not rospy.is_shutdown():

        joint_state_msg.header.stamp = rospy.Time.now()
        joint_publisher.publish(joint_state_msg)
        # this sleep command will pause the program accordin to the rate defined␣
↪above
        rate.sleep()

if __name__ == '__main__':
    try:
        #publish joint angles by calling the main function
        publish_joint_angles()
    except rospy.ROSInterruptException: pass
```

3. Remember to make the script an executable by using the `chmod` command.

4. Now let's create a new launch file to launch the RVIZ simulator, robot_state_publisher. We can modify the launch file that we used in the previous tutorial. Simply remove the commands to launch the `robot_joint_publisher` package. Instead, we are going to run our script to publish joint angles. Create a new launch file with the name `joint_publisher.launch`.

```
<launch>
    <!-- values passed by command line input -->
    <arg name="model" />


  <!-- set these parameters on Parameter Server -->
  <param name="robot_description"
  textfile="$(find serial_link_robot)/urdf/$(arg model)"
  />


  <!-- Start 2 nodes:
    robot_state_publisher and rviz -->



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


  <node name="rviz" pkg="rviz" type="rviz"
    args="-d $(find serial_link_robot)/urdf.rviz"
    required="true" />
  </launch>
```

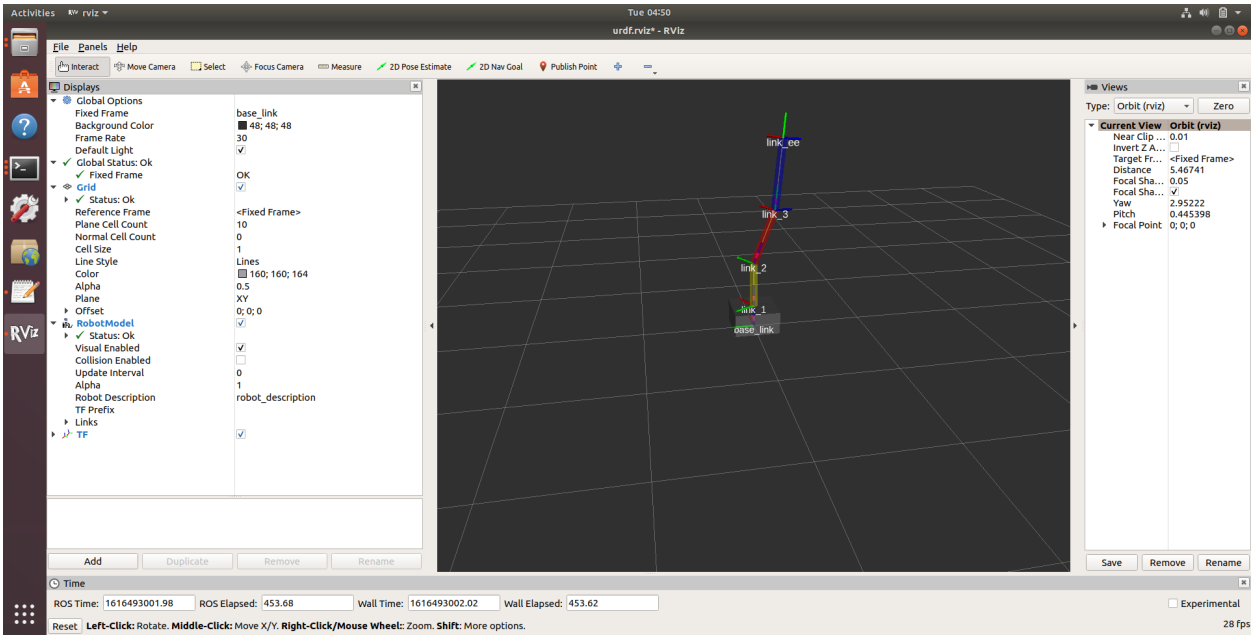
5. Now you can run the launch file by running the roslaunch command in the terminal.

   roslaunch serial_link_robot joint_publisher.launch model:=serial_link_robot.urdf

6. Rviz will show some error messages as joint angles are not yet publishing.

7. Now let's run the script that we created in a new terminal. This will publish the joint angles that we coded in our script. In RVIZ you will see now that the robot has moved to the positions that we commanded using our script.

```
rosrun serial_link_robot joint_publisher.py
```



1. Now try to understand the following script that publishes random joint angles. It will make our robot to do some dance moves.

```
#!/usr/bin/env python
import rospy

import math

from sensor_msgs.msg import JointState
from std_msgs.msg import Header

# generate random floating point values
from random import seed
from random import random
# seed random number generator
seed(1)


def publish_joint_angles():

    rospy.init_node('robot_joint_publisher', anonymous=True)
    joint_publisher =rospy.Publisher('/joint_states',JointState,queue_size=10)
    joint_state_msg = JointState()

    rate = rospy.Rate(10) # 10hz

    joint_state_msg.header = Header()
    joint_state_msg.name = ['joint_1', 'joint_2', 'joint_3']
    # angles are in radians
    joint_state_msg.position = [random()*2*math.pi, random()*2*math.pi,␣
↪random()*2*math.pi]
    joint_state_msg.velocity = []
    joint_state_msg.effort = []
```

(continues on next page)

(continued from previous page)

```
    while not rospy.is_shutdown():

       joint_state_msg.header.stamp = rospy.Time.now()
       joint_state_msg.position = [random(), random(), random()]
       joint_publisher.publish(joint_state_msg)
       rate.sleep()

if __name__ == '__main__':
    try:
        #publish joint angles
        publish_joint_angles()
    except rospy.ROSInterruptException: pass
```

CHAPTER

NINE

ENEX13004:WEEK 3- ROBOTIC ARM COORDINATE TRANSFORMATION

9.1 Transforming End effector position to the base_link

In this exercise, we are going to write a python code to find the position of the End effector w.r.t to the base_link. This is a part of the assignmnet 1 , Q2 requirement. I am not going to provide the complete code but I will provide you with the code skeleton. You need to add a few extra lines to complete the Q2 requirements.

1. First we need to add extra joint (joint_ee) and a link (link_ee) to assign a coordinate frame to the end-effector position. Rememberto change the `Joint_ee` orgin according to the assignment requirements. Add the following xml code to the urdf file before the `</robot>` tag.

```
<!-- Joint ee -->
 <joint name="joint_ee" type="fixed">
    <parent link="link_3"/>
    <child link="link_ee"/>
    <origin xyz="0 0.6 0" rpy="0 0 0" />

  </joint>

<!-- Link ee -->
<link name="link_ee">

  </link>
```

2. You can use the following code skeleton to do the transformation from the `link_ee` to `base_link`. Look at the transformation listner tutorial that we did in week 2. Complete the missing commands.

```
#!/usr/bin/env python
import rospy

import math
import tf
import geometry_msgs.msg



def tf_listner():

    rospy.init_node('robot_tf_listner')
    listener = tf.TransformListener()
```

(continues on next page)

(continued from previous page)

```
    rate = rospy.Rate(10) # 10hz


    while not rospy.is_shutdown():

       # complet the missing command to create the Transform listner from link_ee␣
↪to base_link.
       # You need to then print the translation component to the terminal.

        rate.sleep()

if __name__ == '__main__':
    try:
        #publish joint angles
        tf_listner()
    except rospy.ROSInterruptException: pass
```

3. Name the script as `robot_transform.py` and run it (make it execeutable first). Remember to launch the launch file to run the RVIZ and the robot_state_publisher.
4. Watch the live or the recoded class for more information

CHAPTER

TEN

ENEX13004:WEEK 4- MOVE IT - ROBOTIC ARM SIMULATION

10.1 Using MoveIt Package for Robotic Arm Path Planning

In this tutorial we will creat and configure a robotic arm to use the MoveIt ros package. Then we can use RVIZ to move the rootic arm to the desired location

Let's make sure that we have the most up to date packages: ::

```
rosdep update
sudo apt-get update
sudo apt-get dist-upgrade
```

Install MoveIt

The simplest way to install MoveIt is from pre-built binaries (Debian):

```
sudo apt install ros-melodic-moveit
```

10.2 Creating a Moveit package for a robotic arm

Now let's configure the ROS moveit package for our robotic arm. Then we can use RVIZ to set goal locations and the moveit package will do the inverse kinematics and the path planning.

- First download and save the URDF file that we created in last week to the home folder. You can download the urdf file from the moodle site.

10.2.1 MoveIt Setup Assistant

The MoveIt Setup Assistant is a graphical user interface for configuring any robot for use with MoveIt. Its primary function is generating a Semantic Robot Description Format (SRDF) file for your robot. Additionally, it generates other necessary configuration files for use with the MoveIt pipeline. To learn more about the SRDF, you can go through the URDF/SRDF Overview page.

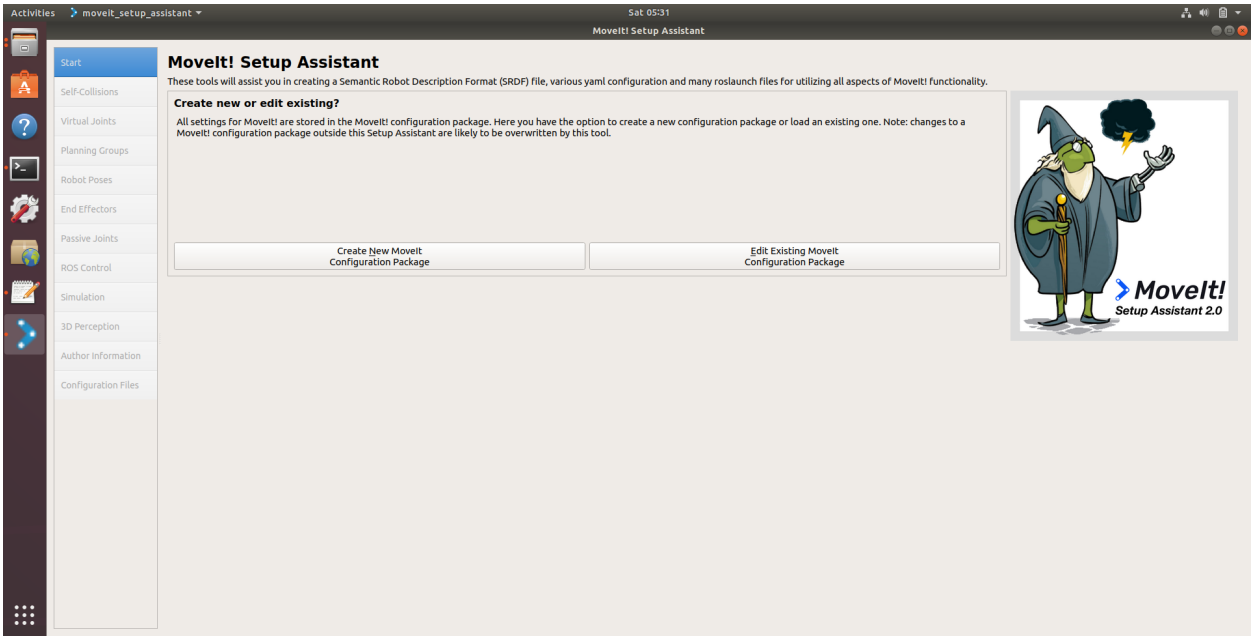
Step 1: Start

To start the MoveIt Setup Assistant:

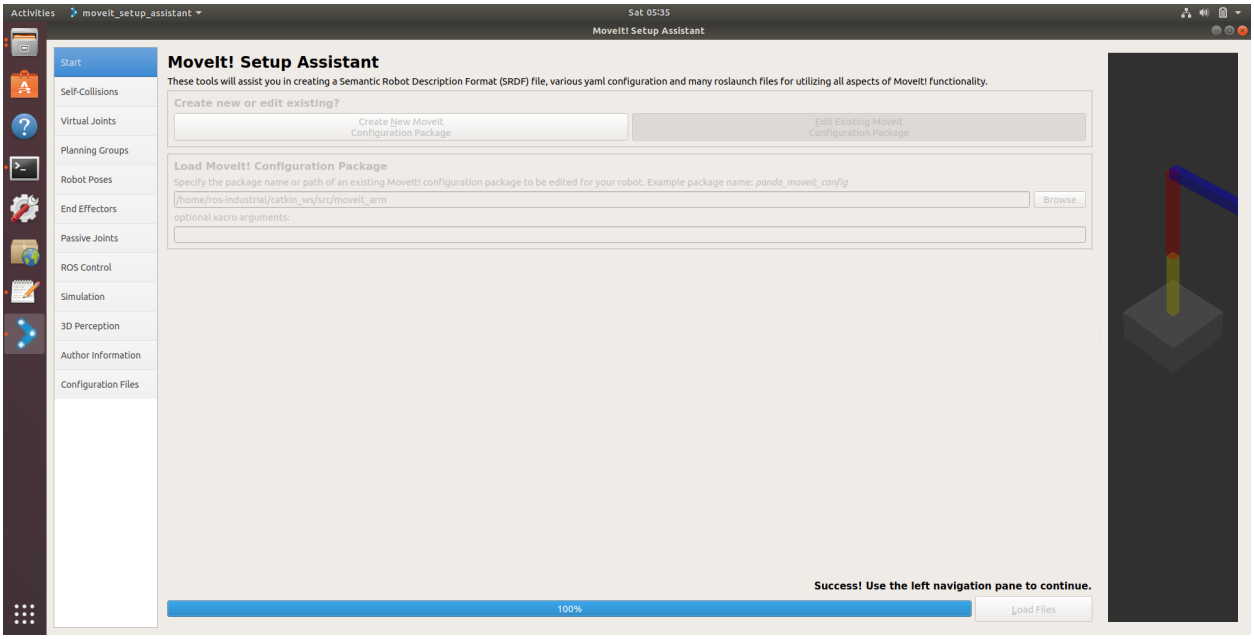
```
roslaunch moveit_setup_assistant setup_assistant.launch
```

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 screens.:



Click on the browse button and navigate to the home folder and choose the URDF file of your robotic arm and then click Load Files. The Setup Assistant will load the files (this might take a few minutes)

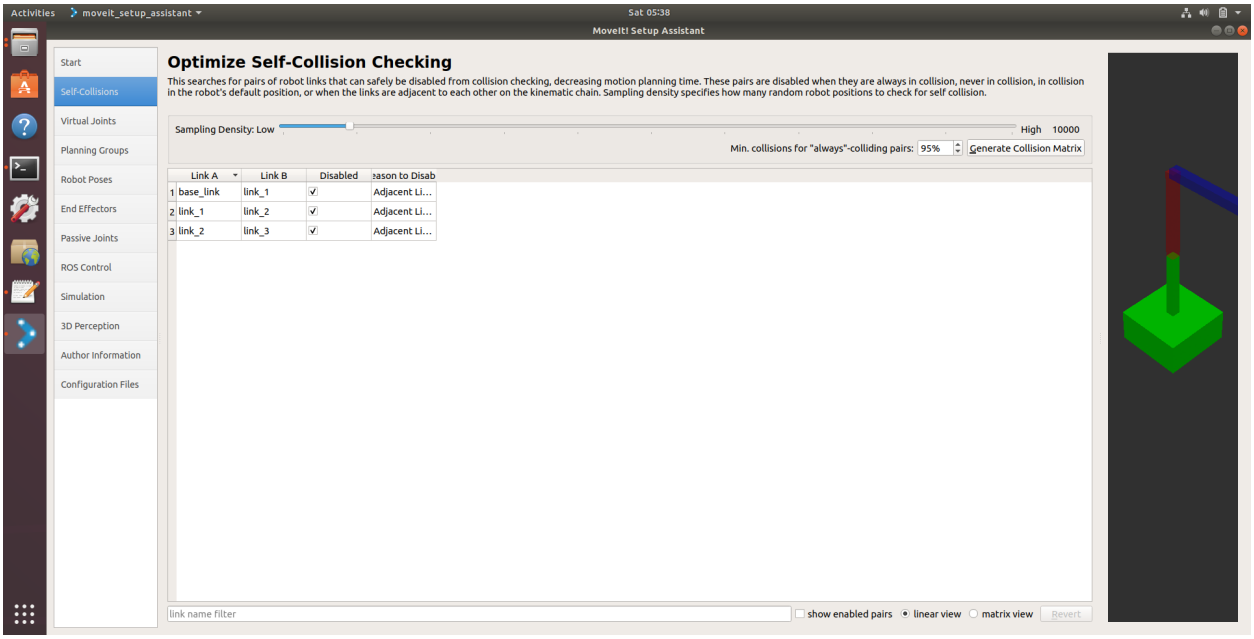


Step 2: Generate Self-Collision Matrix

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. 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.

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.

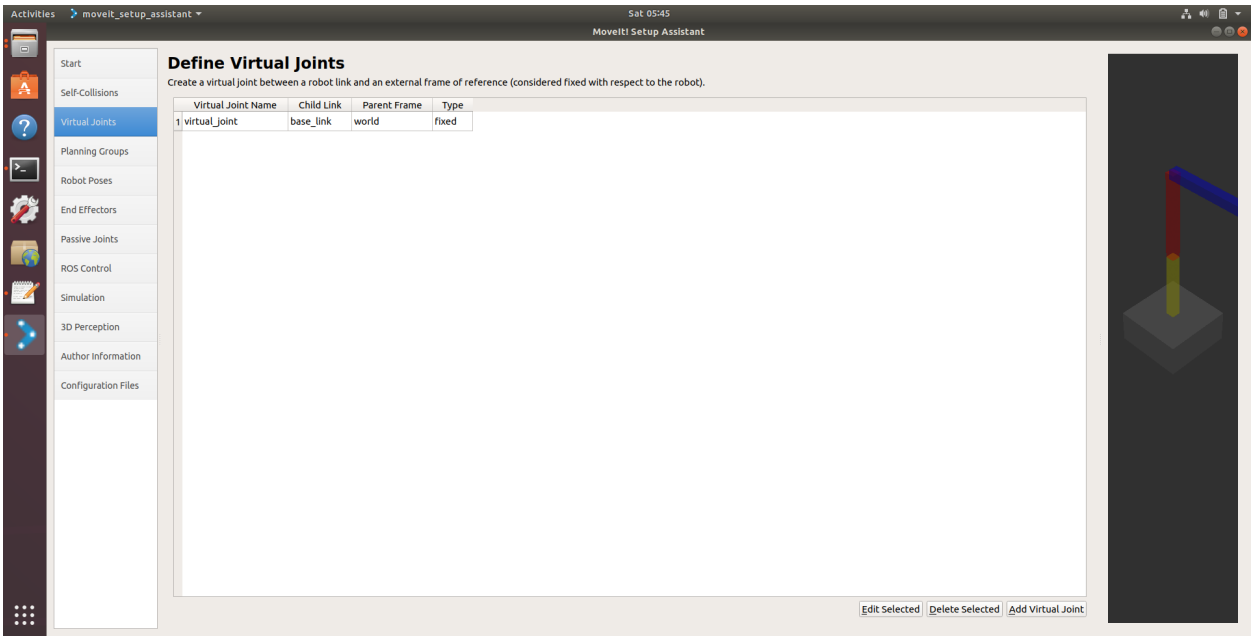


Srep 3 : Virtual Joints

Virtual joints are used primarily to attach the robot to the world. For our robot modela we will define only one virtual joint attaching the base_link of the our robotic arm 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”.

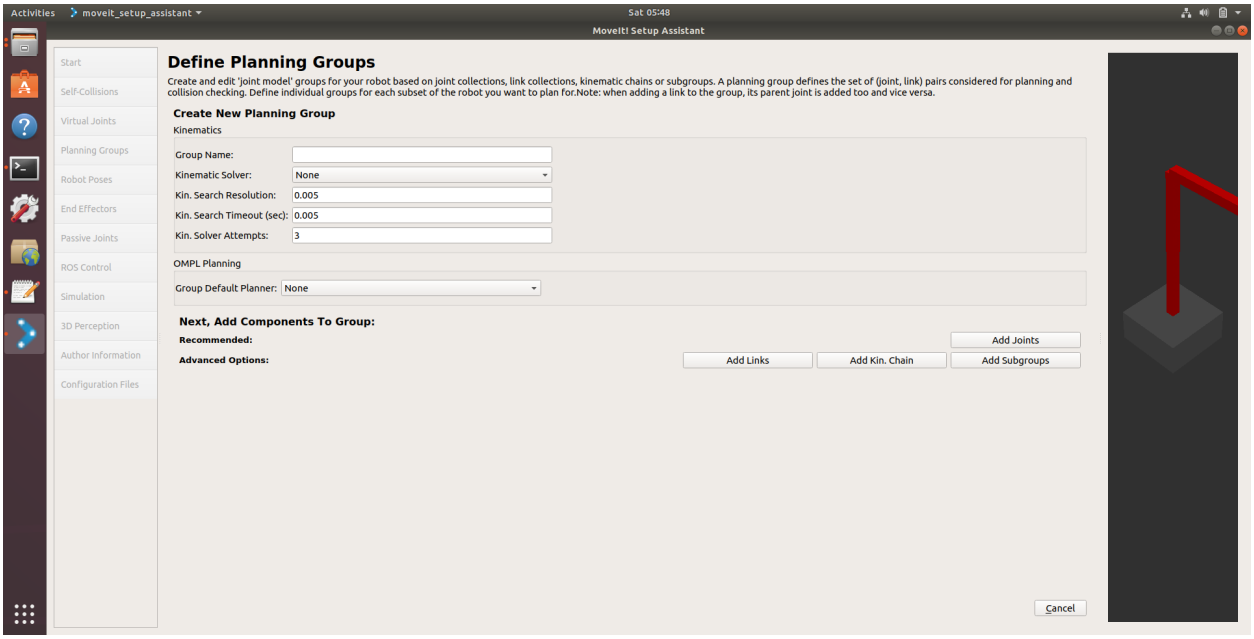
Click Save and you should see this screen:



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.

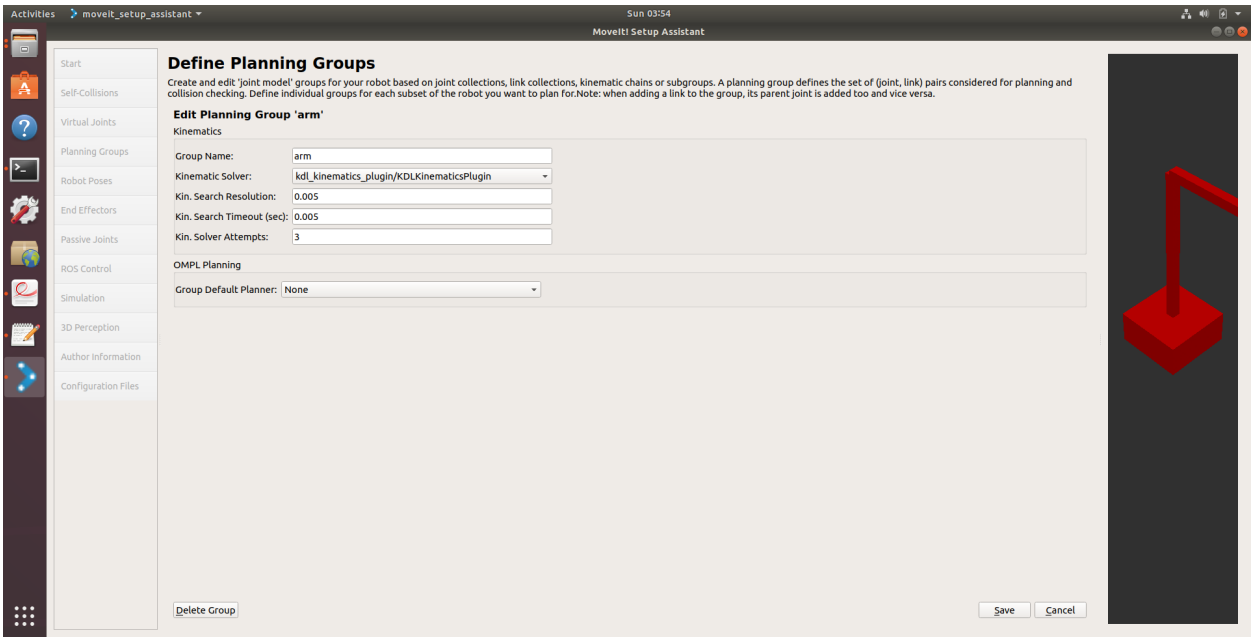
- Click on the Planning Groups pane selector.
- Click on Add Group and you should see the following screen:



Add the arm

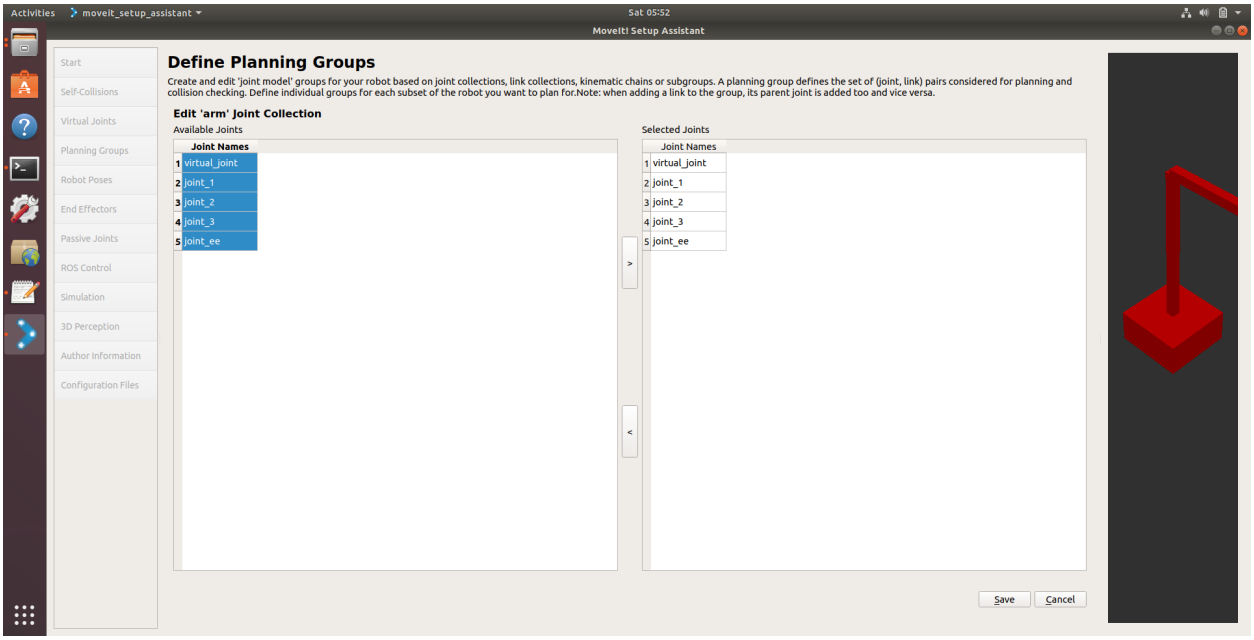
We will first add as a planning group

- Enter Group Name as `arm`
- Choose kdl_kinematics_plugin/KDLKinematicsPlugin as the kinematics solver.
- et Kin. Search Resolution and Kin. Search Timeout stay at their default values.

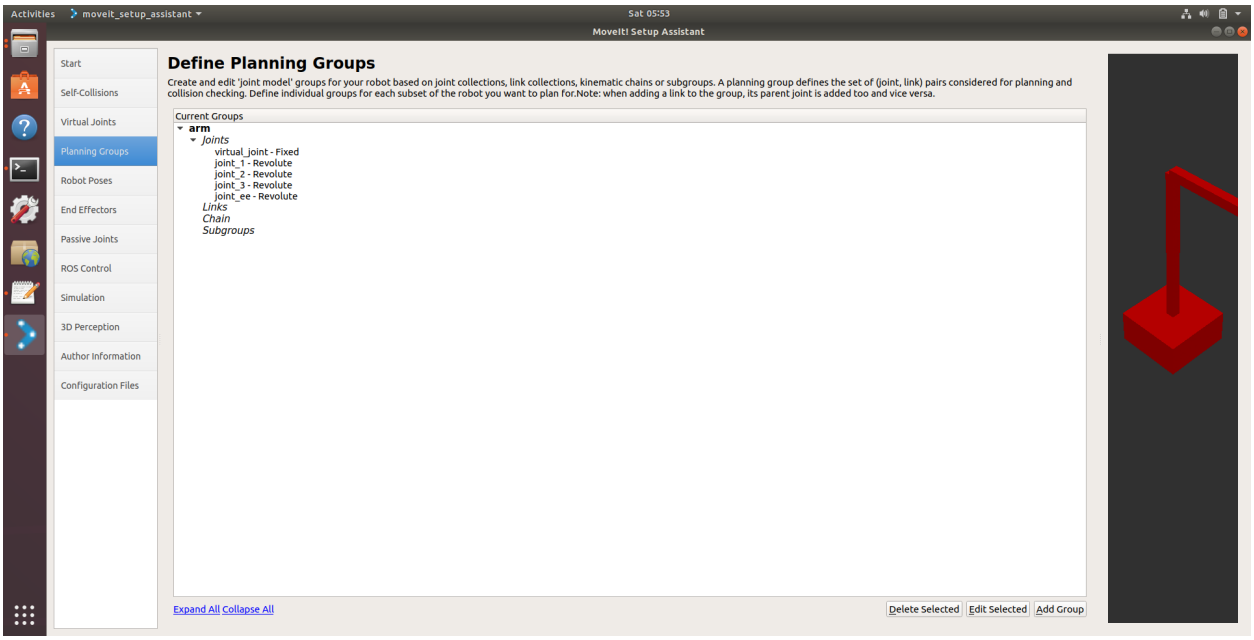


Now, click on the Add Joints button. You will see a list of joints on the left hand side. You need to choose all the joints that belong to the arm and add them to the right hand side. The joints are arranged in the order that they are stored in an internal tree structure. This makes it easy to select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

Click on `virtual_joint`, hold down the Shift button on your keyboard and then click on the `joint_ee`. Now click on the > button to add these joints into the list of selected joints on the right.



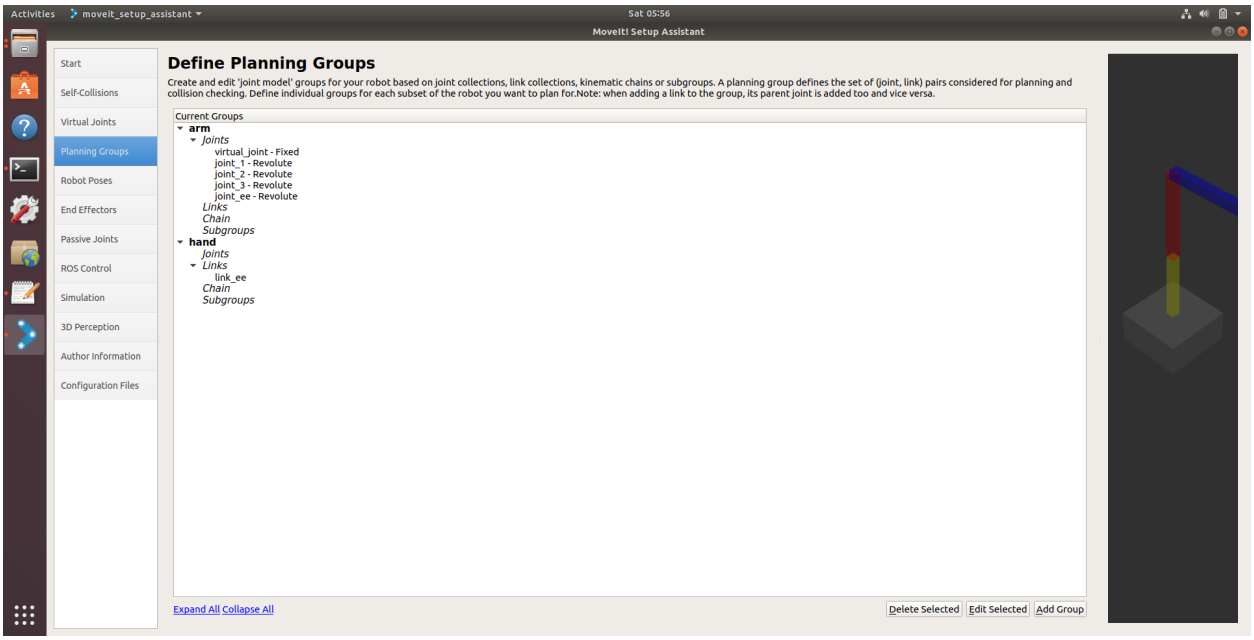
Click Save to save the selected group.



Adding 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.

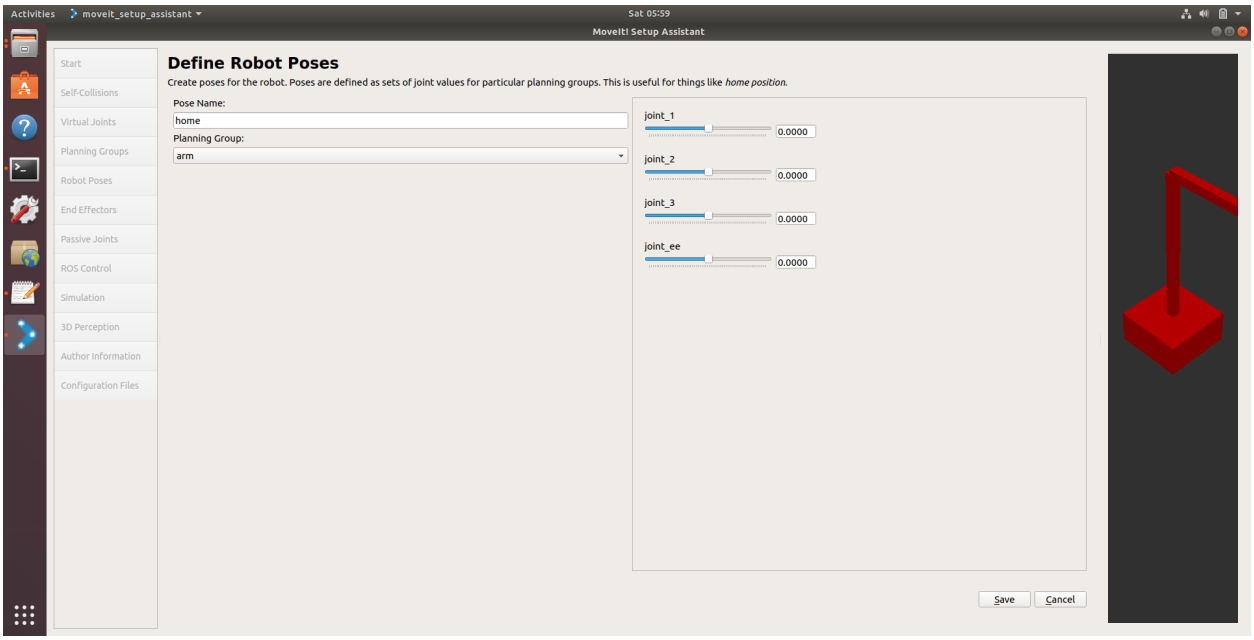
- Click on the Add Group button.
- Enter Group Name as hand
- Let Kin. Search Resolution and Kin. Search Timeout stay at their default values.
- Click on the Add Links button.
- Choose link_ee and add them to the list of Selected Links on the right hand side.
- Click Save



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.

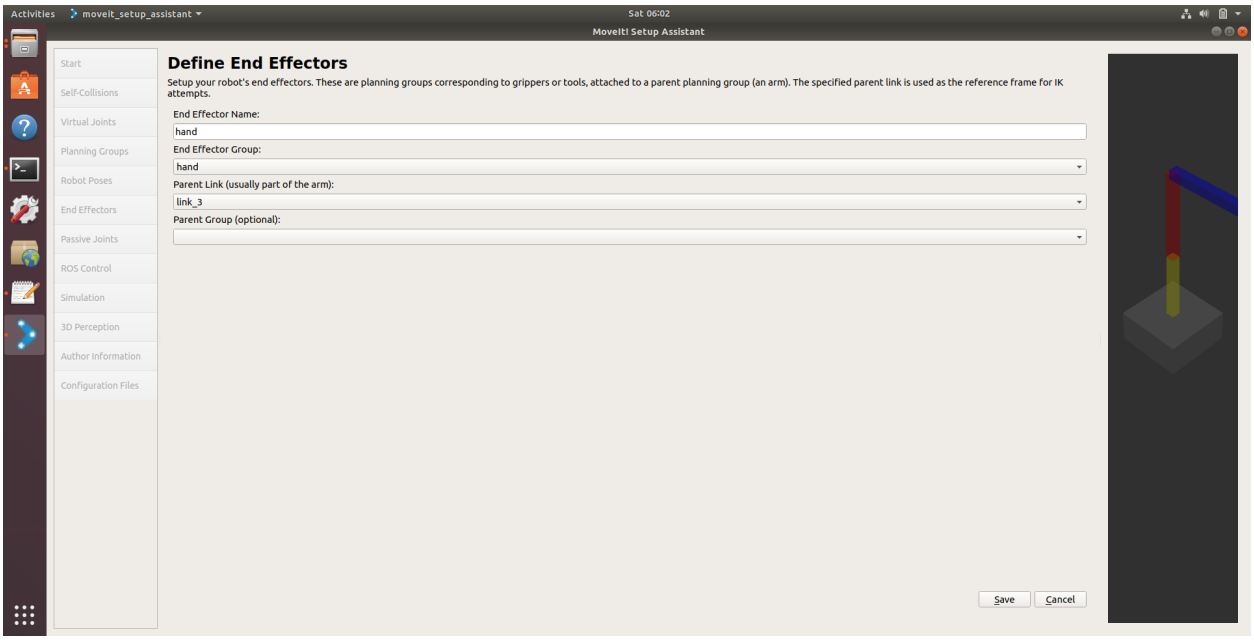
- Click on the Robot Poses pane.
- Click Add Pose. Choose a name for the pose. The robot will be in its Default position where the joint values are set to the mid-range of the allowed joint value range. Move the individual joints around until you are happy and then Save the pose. Note how poses are associated 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



Step 6: Label End Effectors

We have already added the gripper of the robotic arm. 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 link_3 as the Parent Link for this end-effector.
- Leave Parent Group blank.



Step 7: Add Passive Joints

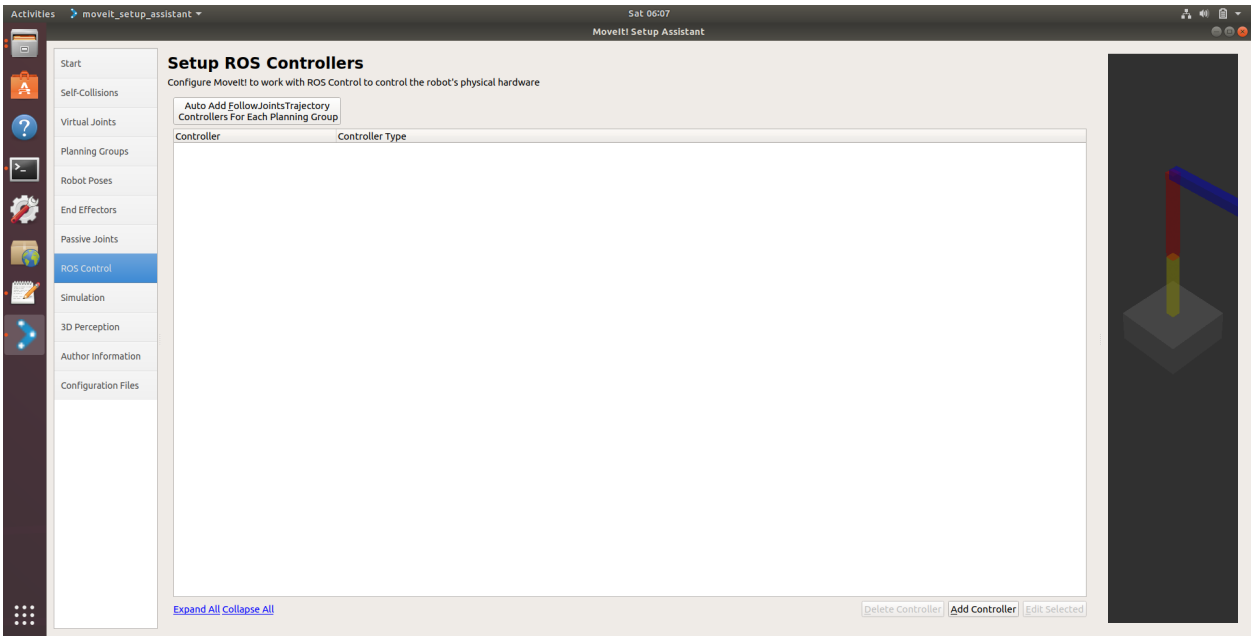
The passive joints tab is meant to allow specification of any passive joints that might exist in a robot. These are joints that are unactuated on a robot (e.g. passive casters.) This tells the planners that they cannot (kinematically) plan for these joints because they can't be directly controlled. Our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

Step 8: 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

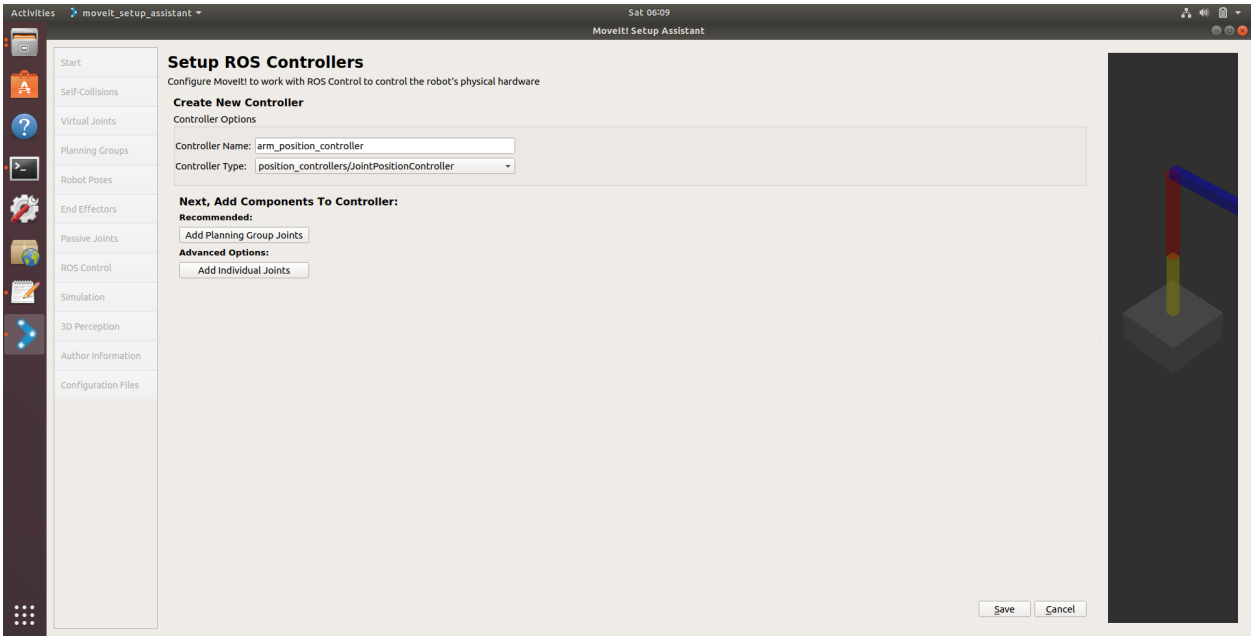
ROS Control tab can be used to auto generate simulated controllers to actuate the joints of the robot. This will allow us to provide the correct ROS interfaces MoveIt.

Click on the ROS Control pane selector.

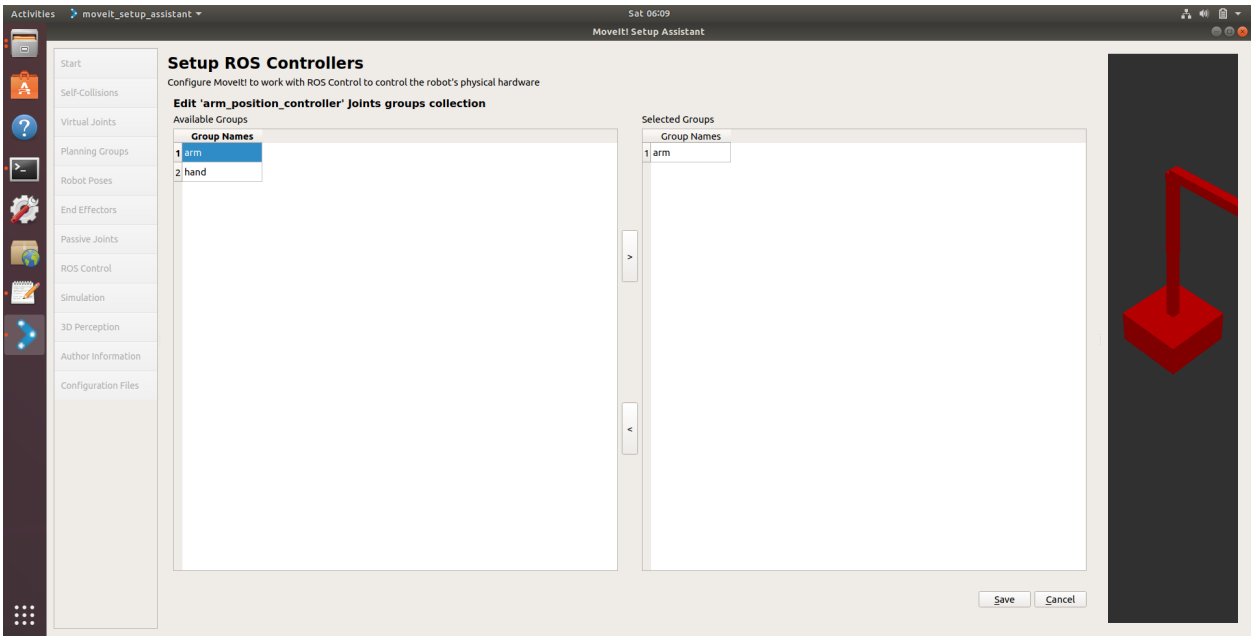


Click on Add Controller and you should see the following screen:

- We will first add the 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.



Choose arm planning group to add all the joints in that group to the arm controller and click save



Step 9 - Simulation

We will skip simulation tab. We will do gazebo simulation in a later stage

Step 10 - 3D perception

We will skip 3D perception as we don't have a attached 3D sensor.

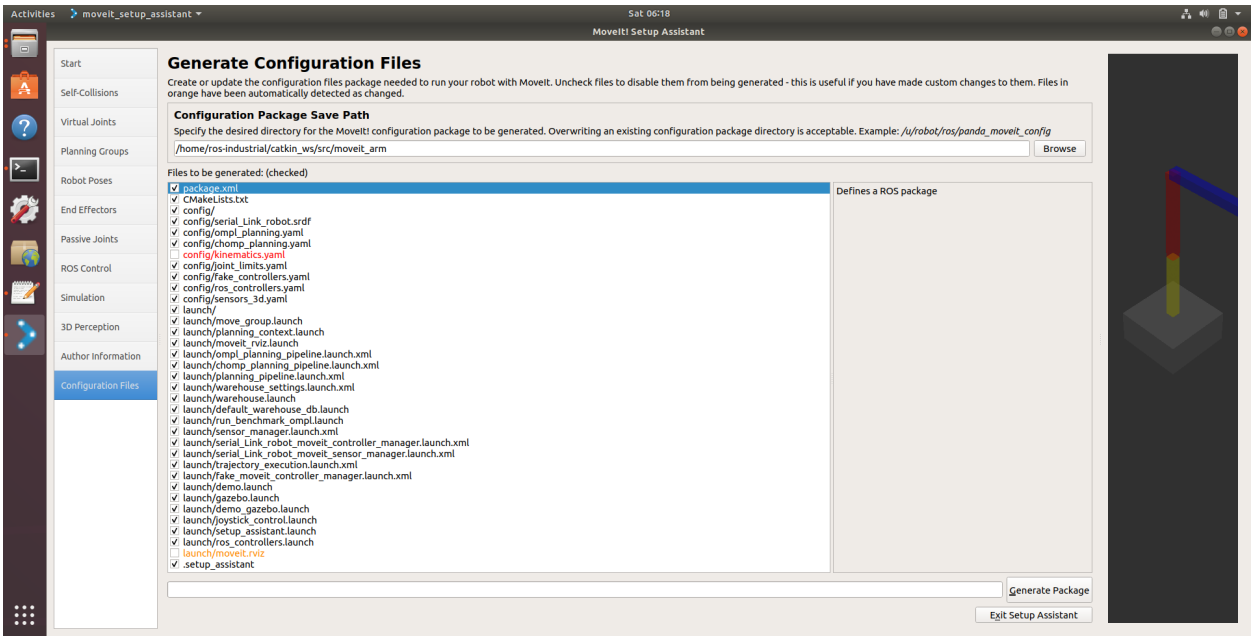
Step 11: Generate Configuration Files

You are almost there. One last step - generating all the configuration files that you will need to start using MoveIt

We need to create a seperate folder inside our catkin_ws folder to save the move_it configuration. Using the UBUNTu GUI click Files icon then go to `catkin_ws` folder and go to the src folder and create a folder called `moveit_arm` .

Now come back to moveit configuration (click on the blue arrow on the left hand side panel) and Click on the Configuration Files pane. Choose folder that we created by clicking on browse.

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.

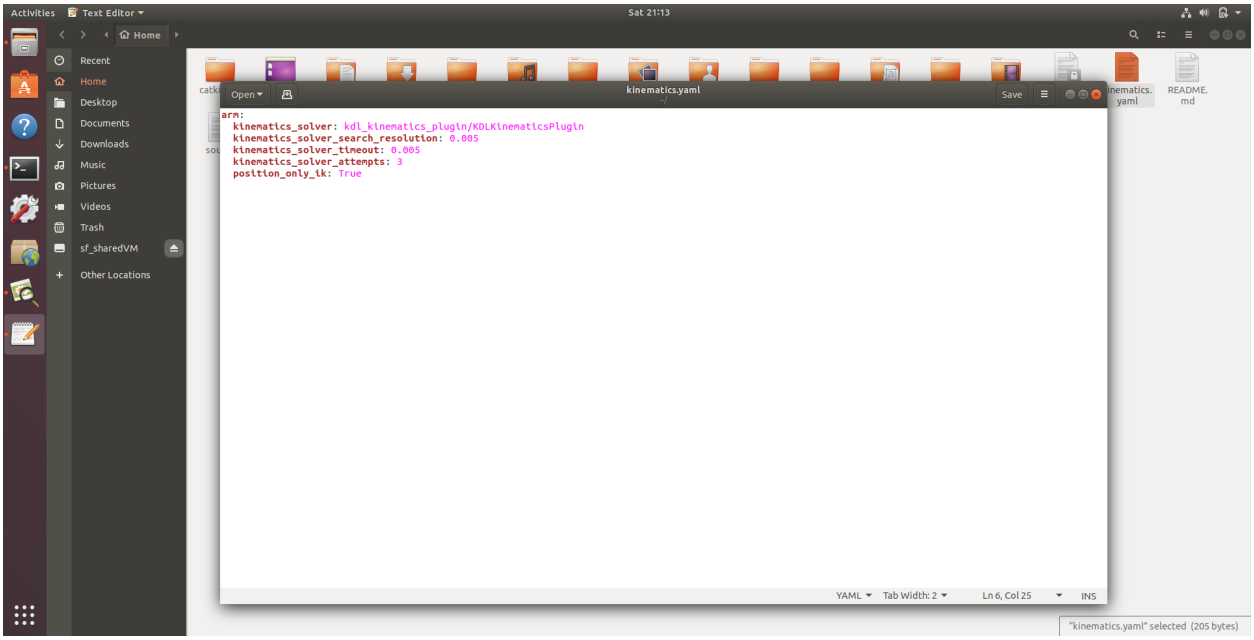


Congratulations!! - You are now done generating the configuration files you need for MoveIt. Now you can exit set up assistance.

Step 12 : Enabling Position onlly Inverse Kinematics

Since our robotic arm has only 3DOF inverse kinematics can't be always solved for 6 DOF pose of the end effector. Therefore we have to enable the position only inverse kinematics manually.

Through the file browser go to `catkin_ws/src/moveit_arm/config` folder and open `kinematics.yaml` file and add `position_only_ik: True` line to the end as shown in the following figure. This will enable the position only kinematics.



License

CHAPTER

ELEVEN

ENEX13004:WEEK 4- MOVE IT - ROBOTIC ARM SIMULATION USING RVIZ

11.1 MoveIt Quickstart in RViz

The quickest way to get started using MoveIt is through its RViz plugin. Rviz is the primary visualizer in ROS and an incredibly useful tool for debugging robotics. The MoveIt Rviz plugin allows you to setup virtual environments (scenes), create start and goal states for the robot interactively, test various motion planners, and visualize the output.

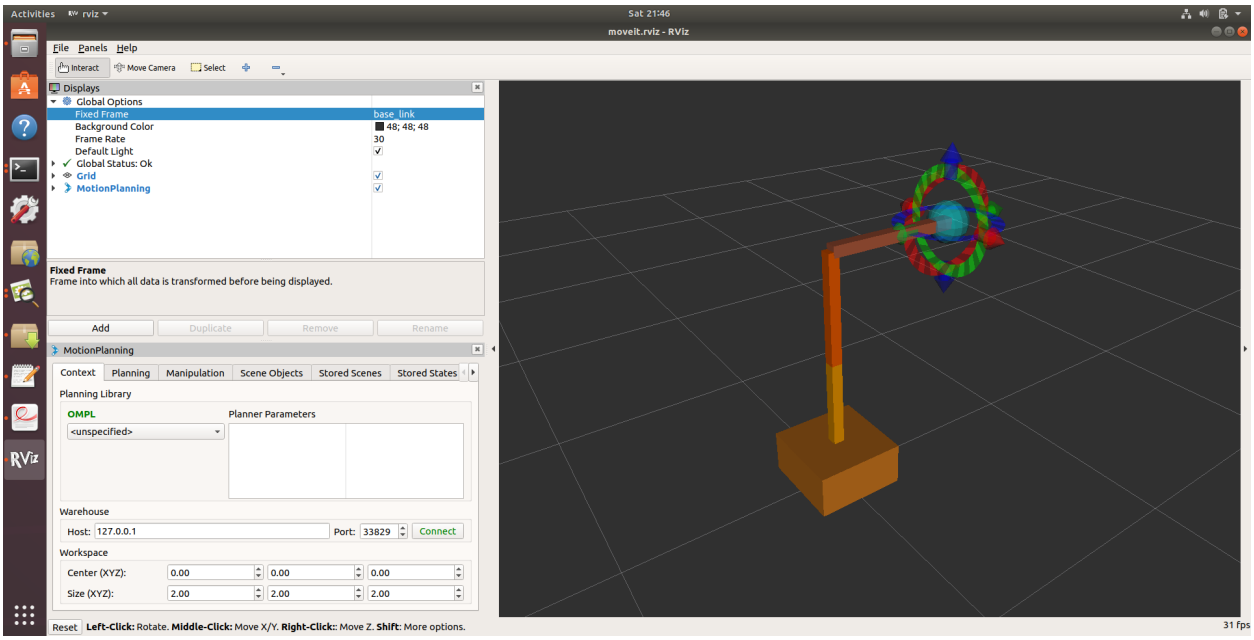
11.1.1 Step 1: Launch the Demo and Configure the Plugin

- First go to your `catKin_ws` and run `catki_make .`
- Also run `rospack profile` command
- Launch the demo: ::

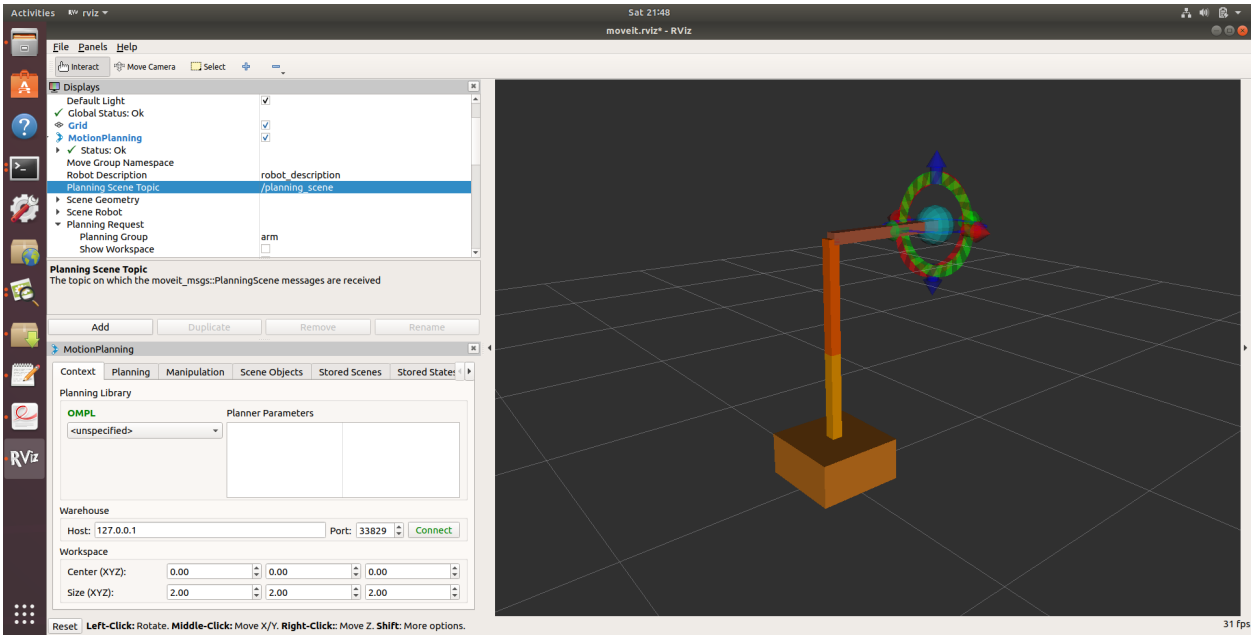
  ```
  roslaunch moveit_arm demo.launch rviz_tutorial:=true
  ```

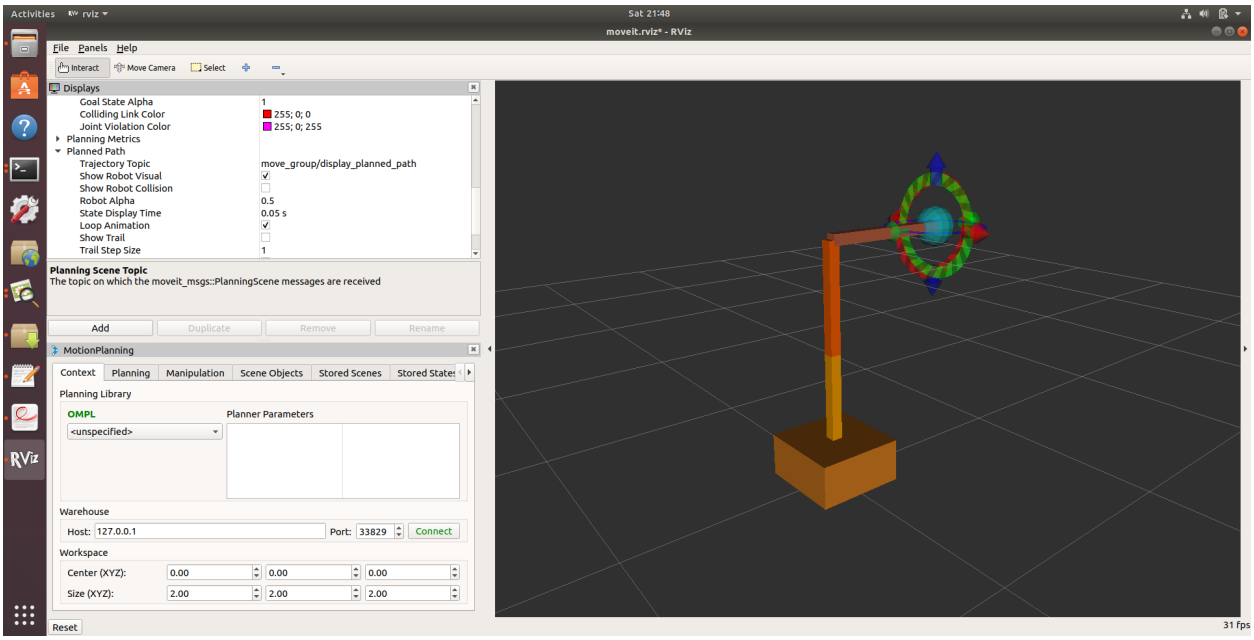
- If you are doing this for the first time, you might see an empty world in RViz and will have to add the Motion Planning Plugin. If you can see the robot then you can skip the following steps
  - You should see an empty world in RViz:
  - In the RViz Displays Tab, press *Add*:
  - From the moveit_ros_visualization folder, choose "MotionPlanning" as the DisplayType. Press "Ok".
  - You should now see the robotic arm in RViz:

  |D|



- Once you have the Motion Planning Plugin loaded, we can configure it. In the "Global Options" tab of the "Displays" subwindow, set the **Fixed Frame** field to `/base_link`
- Now, you can start configuring the Plugin for your robot . Click on "MotionPlanning" within "Displays".
  - Make sure the **Robot Description** field is set to `robot_description`.
  - Make sure the **Planning Scene Topic** field is set to `/planning_scene`. Click on topic name to expose topic-name drop-down.
  - In **Planning Request**, change the **Planning Group** to `arm`.
  - In **Planned Path**, change the **Trajectory Topic** to `/move_group/display_planned_path`.





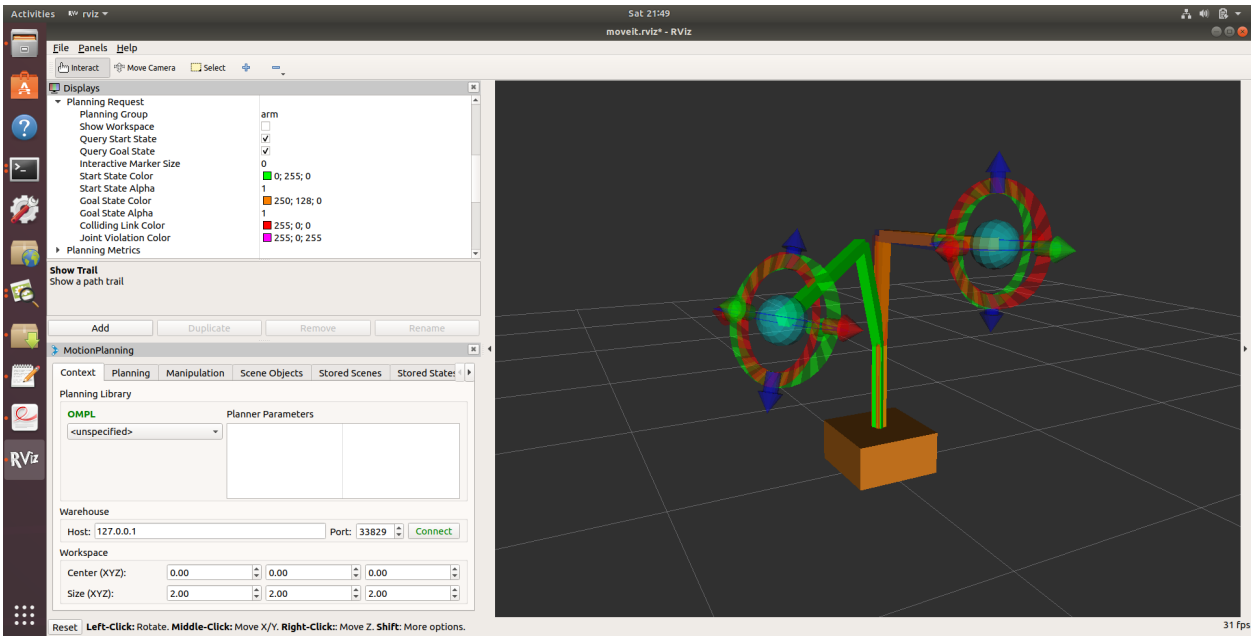
11.1.2 Step 2: Play with the Visualized Robots

There are four different overlapping visualizations:

- The robot's configuration in the `/planning scene` planning environment (active by default).
- The planned path for the robot (active by default).
- Green: The start state for motion planning (disabled by default).
- Orange: The goal state for motion planning (active by default).

The display states for each of these visualizations can be toggled on and off using checkboxes:

- The planning scene robot using the **Show Robot Visual** checkbox in the **Scene Robot** tab.
- The planned path using the **Show Robot Visual** checkbox in the **Planned Path** tab.
- The start state using the **Query Start State** checkbox in the **Planning Request** tab.
- The goal state using the **Query Goal State** checkbox in the **Planning Request** tab.
- Play with all these checkboxes to switch on and off different visualizations.



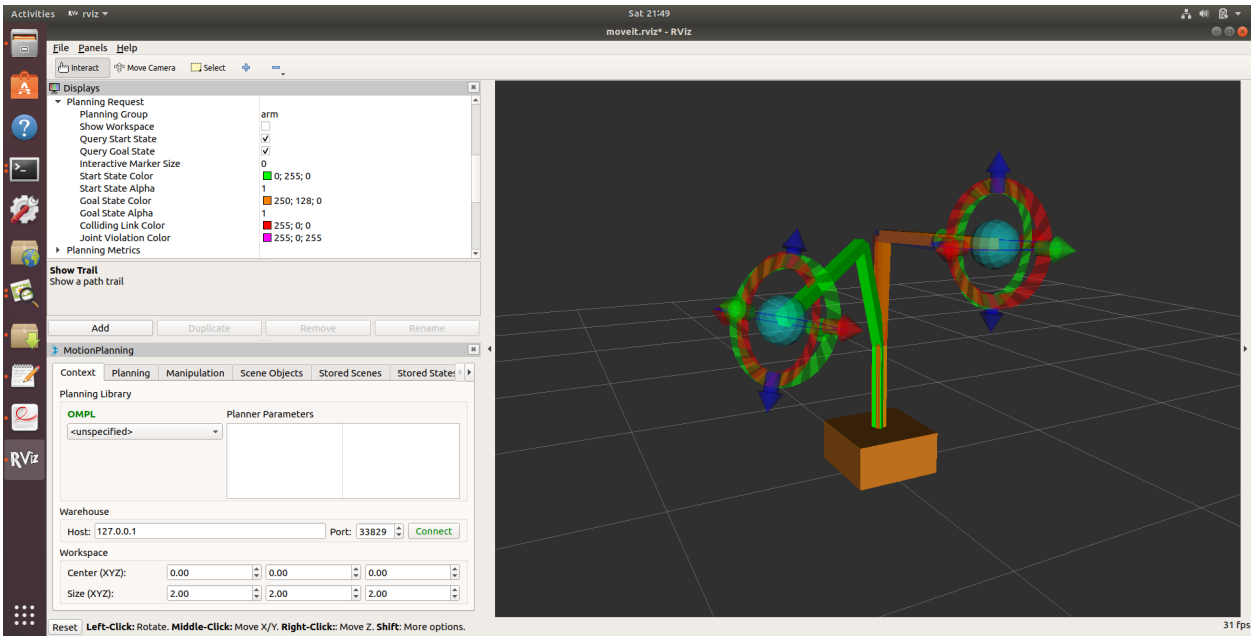
11.1.3 Step 3: Interact with the Robot

For the next steps we will want only the scene robot, start state and goal state:

- Check the **Show Robot Visual** checkbox in the **Planned Path** tab
- Un-check the **Show Robot Visual** checkbox in the **Scene Robot** tab
- Check the **Query Goal State** checkbox in the **Planning Request** tab.
- Check the **Query Start State** checkbox in the **Planning Request** tab.

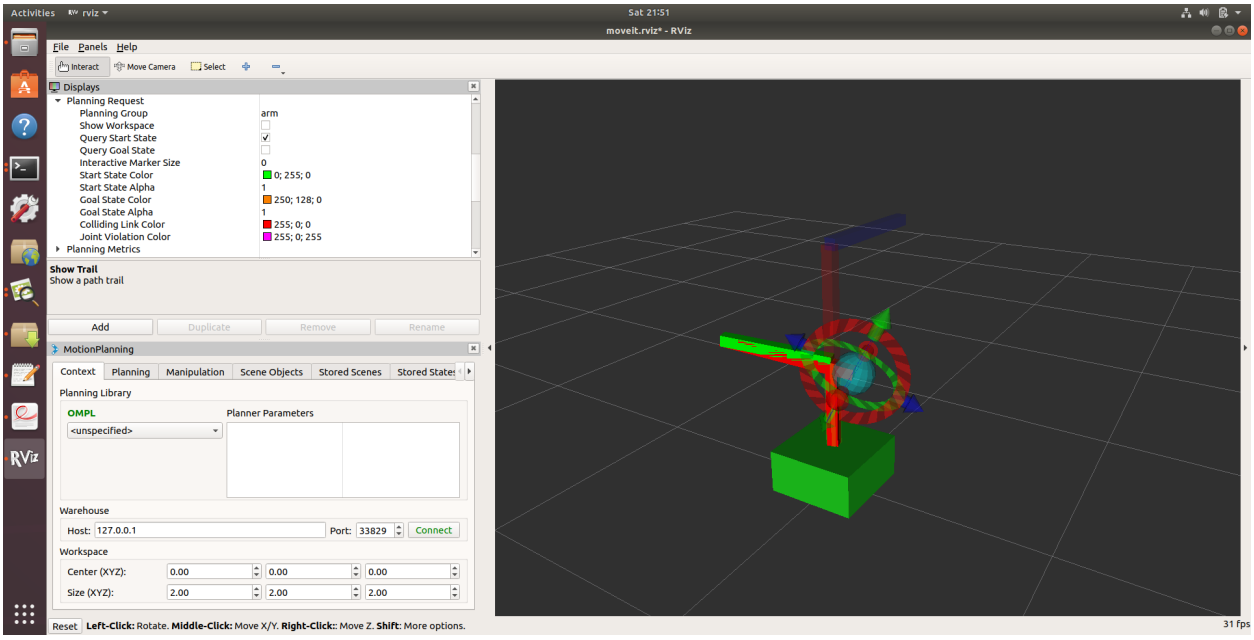
There should now be two interactive markers. One marker corresponding to the orange colored arm will be used to set the "Goal State" for motion planning and the other marker corresponding to a green colored arm are used to set the "Start State" for motion planning. If you don't see the interactive markers press **Interact** in the top menu of RViz (Note: some tools may be hidden, press **"+"** in the top menu to add the **Interact** tool as shown below).

You should now be able to use these markers to drag the arm around and change its orientation. Try it!



11.1.4 Moving into collision

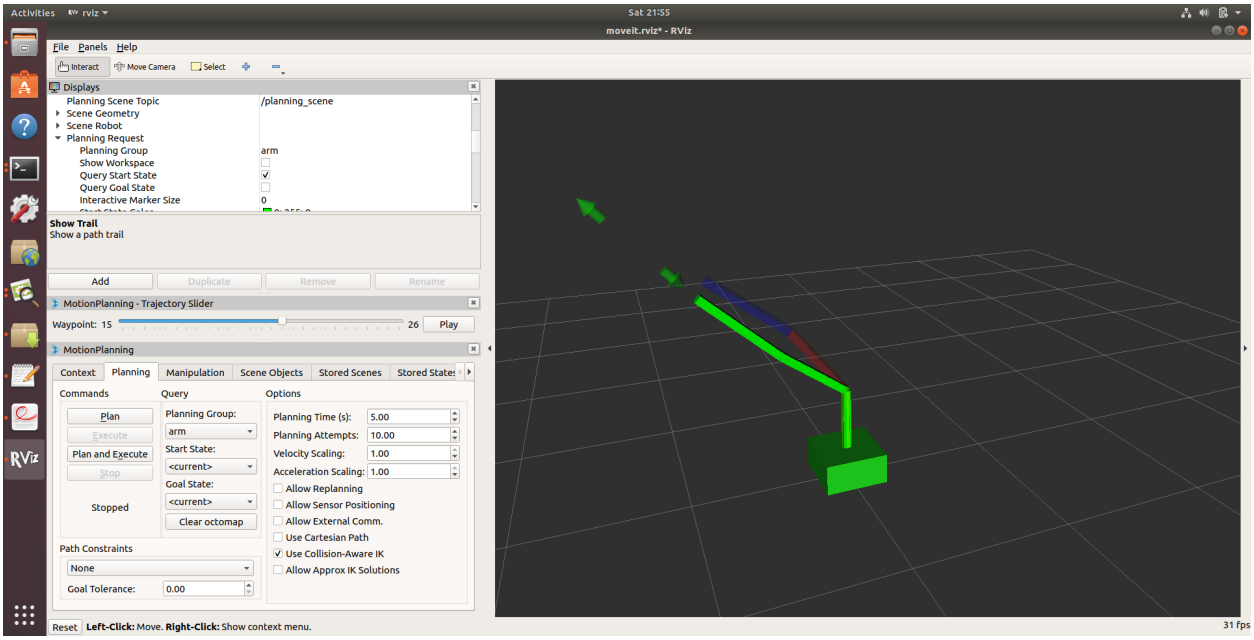
Note what happens when you try to move one of the arms into collision with the other. The two links that are in collision will turn red.



The "Use Collision-Aware IK" checkbox found within the MotionPlanning plugin under the Planning tab allows you to toggle the behavior of the IK solver. When the checkbox is ticked, the solver will keep attempting to find a collision-free solution for the desired end-effector pose. When it is not checked, the solver will allow collisions to happen in the solution. The links in collision will always still be visualized in red, regardless of the state of the checkbox.

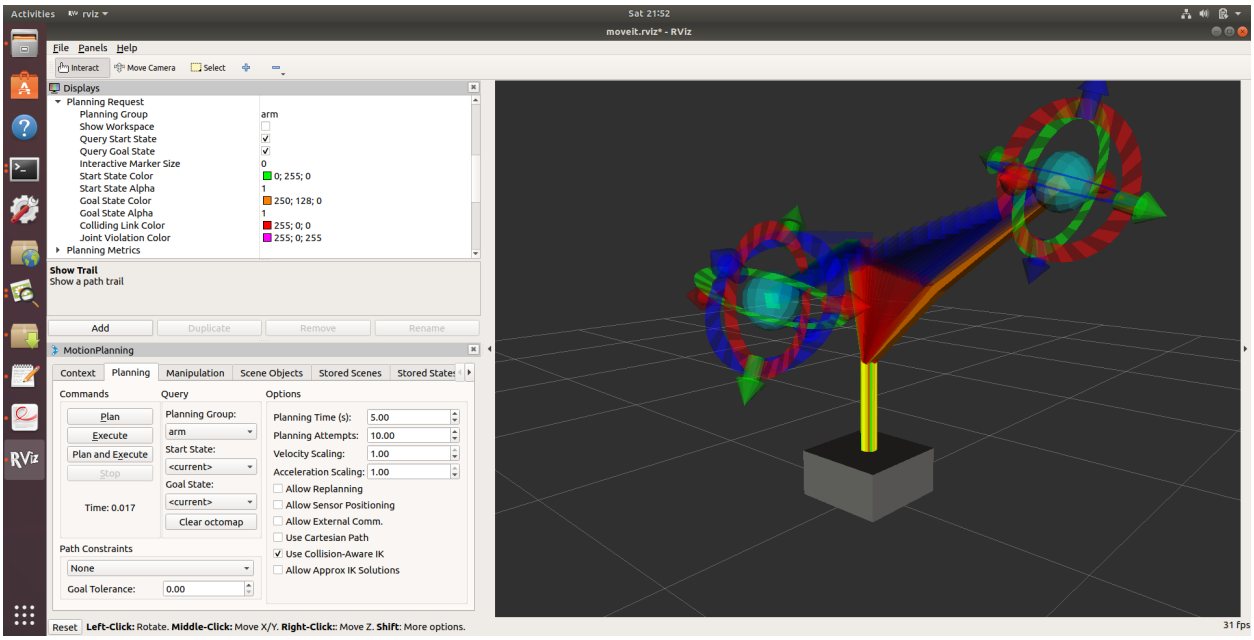
11.1.5 Moving out of Reachable Workspace

Note what happens when you try to move an end-effector out of its reachable workspace.



11.1.6 Step 4: Use Motion Planning with the Robot

- Now, you can start motion planning with the Robtic arm in the MoveIt RViz Plugin.
  - Move the Start State to a desired location.
  - Move the Goal State to another desired location.
  - Make sure both states are not in collision with the robot itself.
  - Make sure the Planned Path is being visualized. Also check the **Show Trail** checkbox in the **Planned Path** tab.
  - In the **MotionPlanning** window under the **Planning** tab, press the **Plan** button. You should be able to see a visualization of the arm moving and a trail.

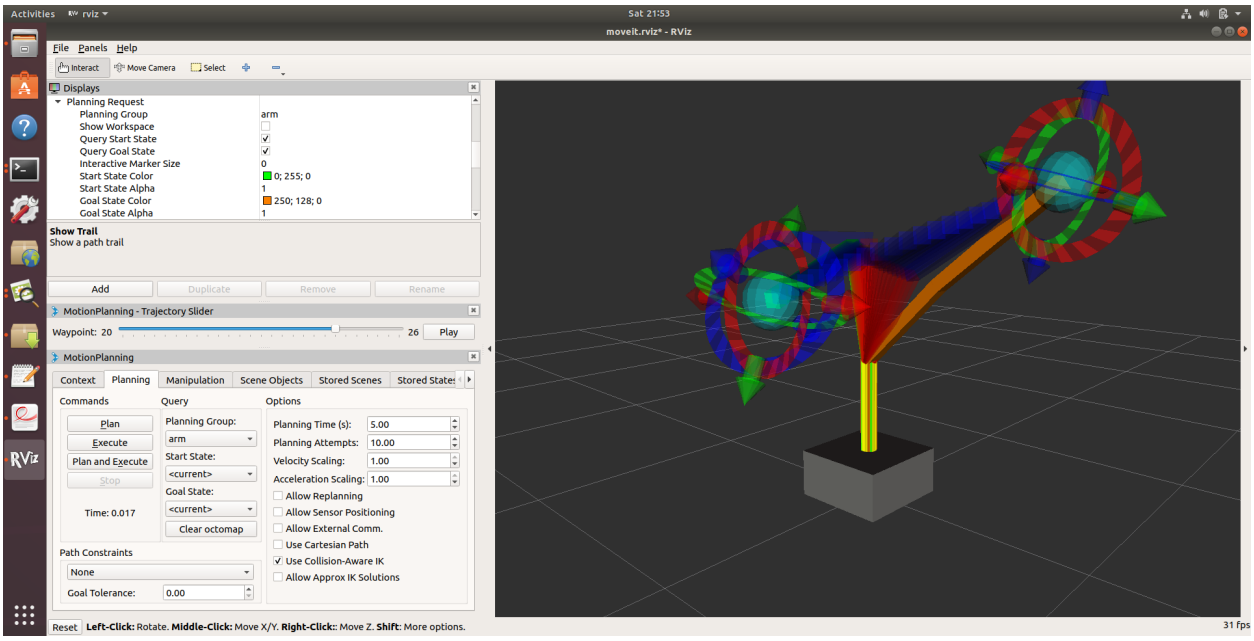


11.1.7 Step 5: Introspecting Trajectory Waypoints

You can visually introspect trajectory point by point on RViz.

- From "`Panels`" menu, select "`MotionPlanning - Slider`". You'll see a new Slider panel on RViz.
- Set your goal pose, then run `Plan`.
- Play with the "`Slider`" panel, e.g. move the slider, push "`Play`" button.

NOTE: Once you placed your EEF to a new goal, be sure to run `Plan` before running `Play` – otherwise you'll see the waypoints for the previous goal if available.



11.1.8 Step 6: Saving Your Configuration

RViz enables you to save your configuration under `File->Save Config`. You should do this before continuing on to the next tutorials.

11.1.9 License

This tutorial is modified from the original content from

http://docs.ros.org/en/melodic/api/moveit_tutorials/html/doc/quickstart_in_rviz/quickstart_in_rviz_tutorial.html

with BSD 3 License.

CHAPTER

TWELVE

ENEX13004:WEEK 4- MOVE IT - ROBOTIC PATH PLANNING WITH PYTHON CODES

12.1 MoveIt Path planning with python code

In this tutorial, we will learn how to write a python code to move our robot arm to a goal location.

One of the simplest MoveIt user interfaces is through the Python-based Move Group Interface. These wrappers provide functionality for most operations that the average user will likely need, specifically setting joint or pose goals, creating motion plans, moving the robot, adding objects into the environment and attaching/detaching objects from the robot.

12.1.1 Creating a python script

Make sure that you have completed the previous MoveIt configuration tutorial. Now go to your `catkin_ws/src/moveit_arm` folder and create an src folder inside it. Then create the `move_robot_arm.py` script inside it.

12.1.2 Python script

- Download `move_robot_arm.py` file from the moodle site and copy it to the python file that you created. See below for more explanation of the code.
- Make the python script executable by using the `chmod` command.
- Build your project by running the catkin_make command inside catkin_ws folder.

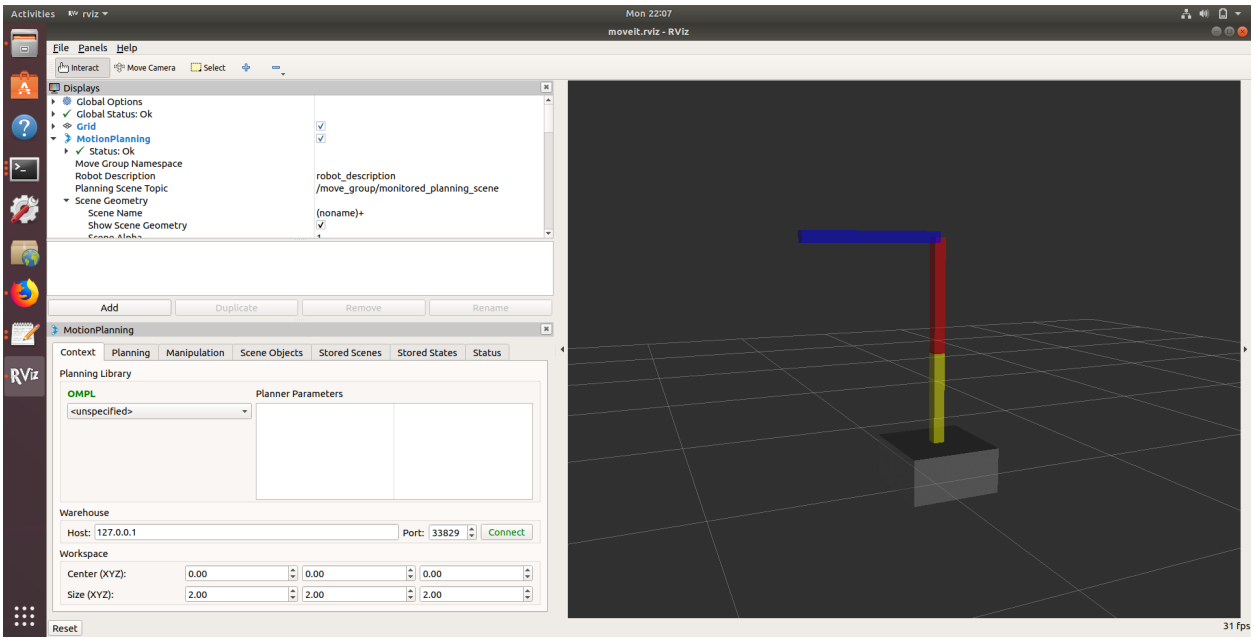
12.1.3 Run and configure RVIZ

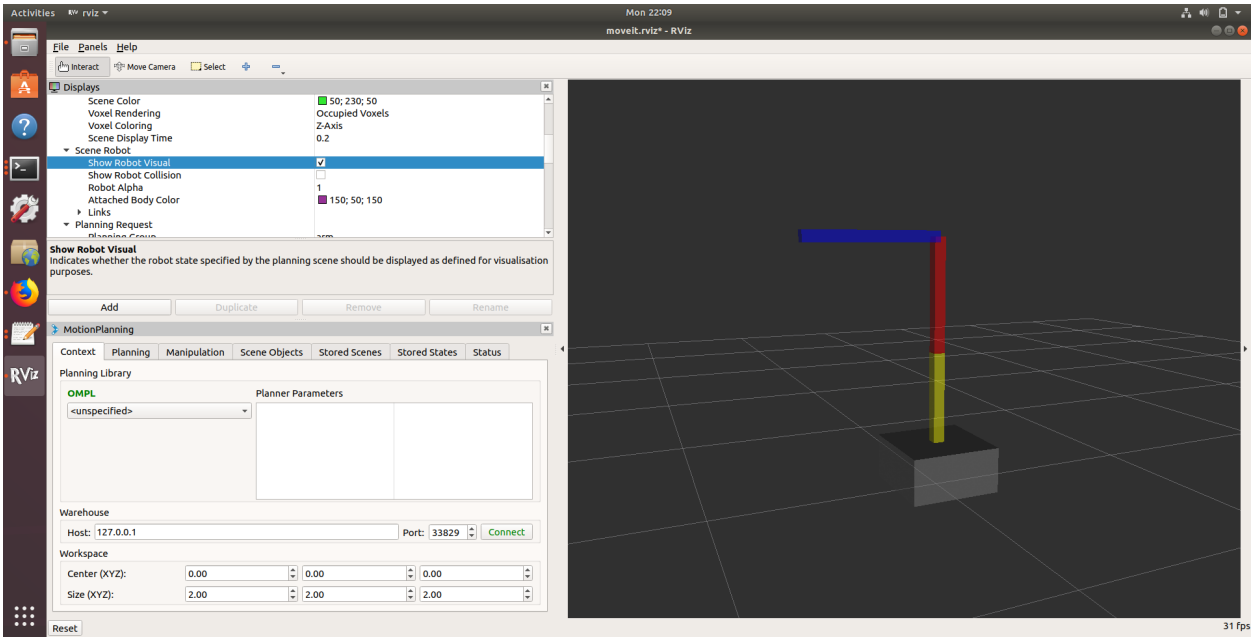
In a new terminal run the follwing command to load RVIZ with moveit motion planning GUI.

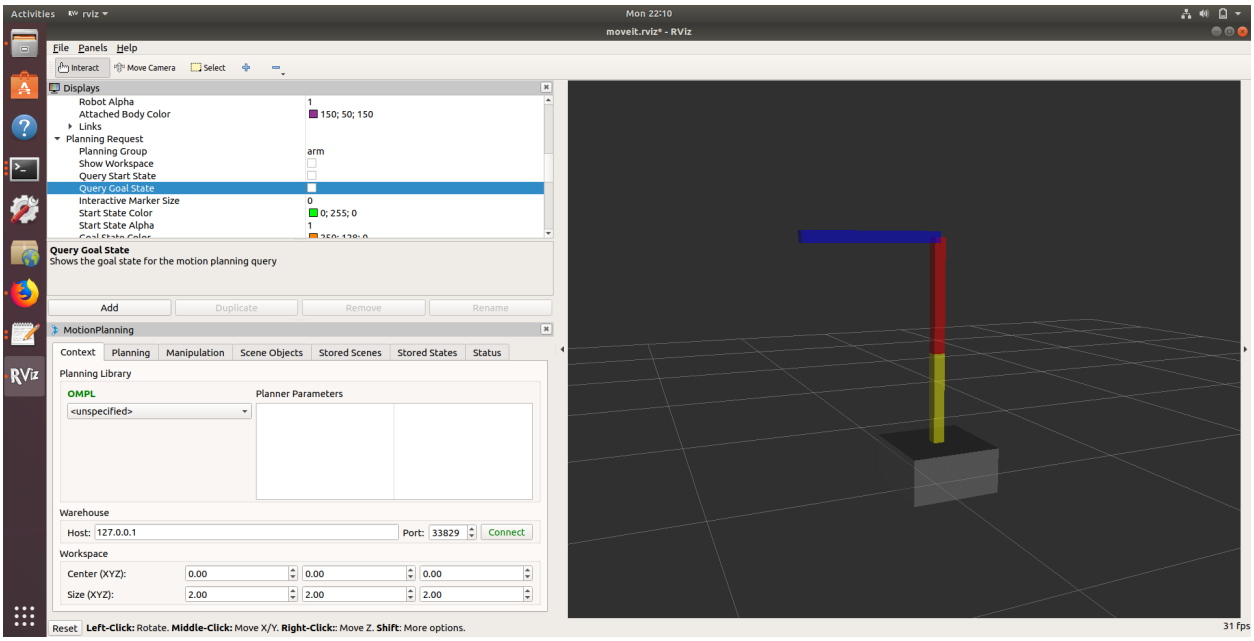
```
roslaunch moveit_arm  demo.launch
```

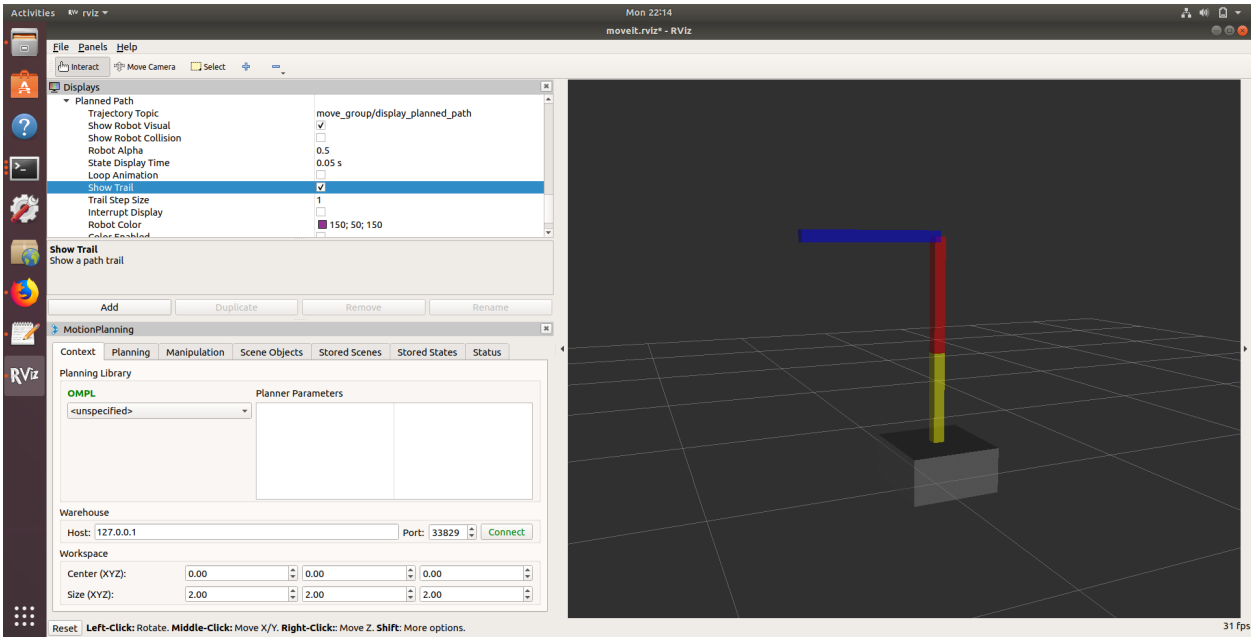
Set following setting in RVIZ

- In motion planning panel set `Planning Scene Topic` to /move_group/monitored_planning_scene.
- In Scene Robot tick `Show Robot Visual`
- In Planning request untick Query `Start State` and `Query Goal State`
- In Planned path tick `Show Trail`
- In Planned path untick `Loop animation`
- save the config by File -> save


Activities
rviz
Mon 22:07
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
robot_description
Planning Scene Topic
/move_group/monitored_planning_scene
Scene Geometry
Scene Name
(noname)+
Show Scene Geometry
Add
Duplicate
Remove
Rename
MotionPlanning
Context
Planning
Manipulation
Scene Objects
Stored Scenes
Stored States
Status
Planning Library
OMPL
Planner Parameters
<unspecified>
Warehouse
Host: 127.0.0.1
Port: 33829
Connect
Workspace
Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00
Reset
31 fps


Activities
rviz
Mon 22:09
moveit.rviz* - RViz
File Panels Help
Interact
Move Camera
Select
Displays
Scene Color
50; 230; 50
Voxel Rendering
Occupied Voxels
Voxel Coloring
Z-Axis
Scene Display Time
0.2
Scene Robot
Show Robot Visual
Show Robot Collision
Robot Alpha
1
Attached Body Color
150; 50; 150
Links
Planning Request
Show Robot Visual
Indicates whether the robot state specified by the planning scene should be displayed as defined for visualisation purposes.
Add
Duplicate
Remove
Rename
MotionPlanning
Context
Planning
Manipulation
Scene Objects
Stored Scenes
Stored States
Status
Planning Library
OMPL
Planner Parameters
<unspecified>
Warehouse
Host: 127.0.0.1
Port: 33829
Connect
Workspace
Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00
Reset
31 fps



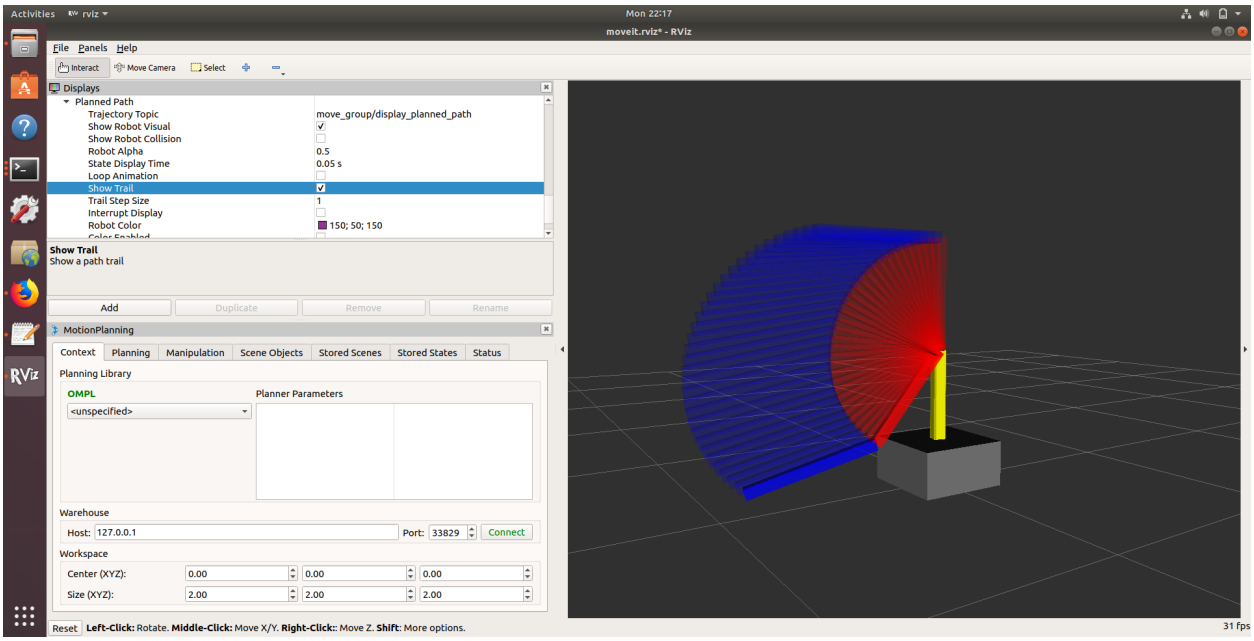


12.1.4 Run path planning script

In a new terminal run

```
rosrun moveit_arm move_robot_arm.py
```

This will ask you to press enter. When you press enter you would see that the arm is moving to the new location. And the path moved is also shown in Rviz.



12.1.5 Setting new goals

We can set a new goal location in the `go_to_pose_goal()` function. Try to change the goal location by changing the x,y,z values in meters w.r.t. `base_link`. See how I have set the goal location. If the goal location cannot be reached then a message will display saying that a valid path cannot be found.

```
# set the new end end effector goal location , w.r.t. base_link in meters
    pose_goal.position.x =  0.3
    pose_goal.position.y = 0.8
    pose_goal.position.z = 0.0
```

12.1.6 Moving on a Cartisian Path

We can also move the robot end effector in a cartesian path (x,y,z). What we have to do is define a set of way points.

Download the `move_robot_arm_path.py` and run in a terminal after copying it into the `moveit_arm\src folder`. Make it executable first. Kill the RVIZ and rerun it in a new terminal.

```
roslaunch moveit_arm  demo.launch

rosrun moveit_arm move_robot_arm_path.py
```

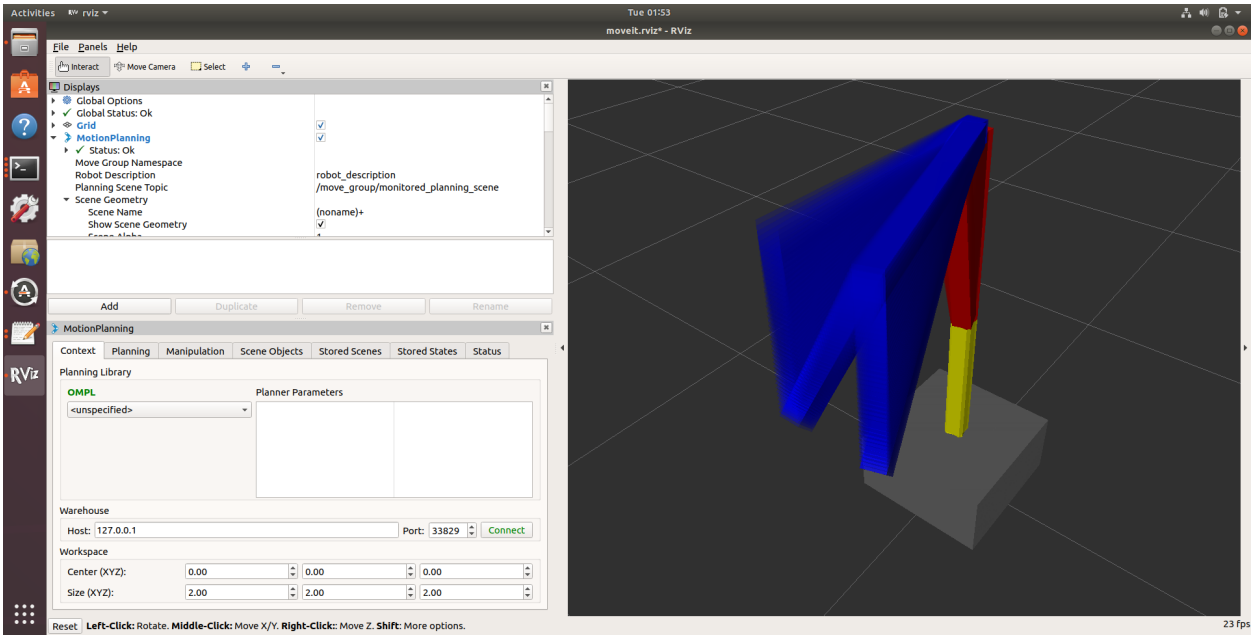
This will move a robot end-effector on a path. Modify the code and see whether you can move the robot's end effector in a path that can draw the first letter of your name. All you need is a set of waypoints. See the code section below where I have added the waypoints. First I record the current location and move the robot's end-effector from the initial position.

```
waypoints = []
```

(continues on next page)

(continued from previous page)

```
    # current position
    wpose = move_group.get_current_pose().pose

    # first way point

    wpose.position.z = wpose.position.z-0.4
    waypoints.append(copy.deepcopy(wpose))

    # second way point
    wpose.position.z = wpose.position.z+0.4
    wpose.position.x = wpose.position.x - 0.2
    waypoints.append(copy.deepcopy(wpose))

    # Third way point
    wpose.position.z = wpose.position.z -0.4
    waypoints.append(copy.deepcopy(wpose))

    # We want the Cartesian path to be interpolated at a resolution of 1 cm
    # which is why we will specify 0.01 as the eef_step in Cartesian
    # translation.  We will disable the jump threshold by setting it to 0.0,
    # ignoring the check for infeasible jumps in joint space, which is sufficient
    # for this tutorial.

    (plan, fraction) = move_group.compute_cartesian_path(waypoints, 0.01, 0.0)
    # now executed the planned path
    move_group.execute(plan, wait=True)
```



Turtle_bot3_simulation.md

CHAPTER

THIRTEEN

ENEX13004:WEEK 5- MOBILE ROBOTS

13.1 Turtlbot 3 simulation using RVIZ

In this tutorial, we will learn how to move Turtlebot3 mobile robot platform in a simulation environment. See this week's lecture for more information about the Turtlebot3 robotic platform.

Then we are also going to run Turtlebot 3 platform in the Gazebo simulation environment. Gazebo is a physics-based simulation environment where we can simulate robots with environmental interactions.

The required packages to simulate Turtlebot 3 (A.K.A Burger) are already installed on your virtual machine.

13.1.1 Launching TurtleBot 3 simulation in RVIZ

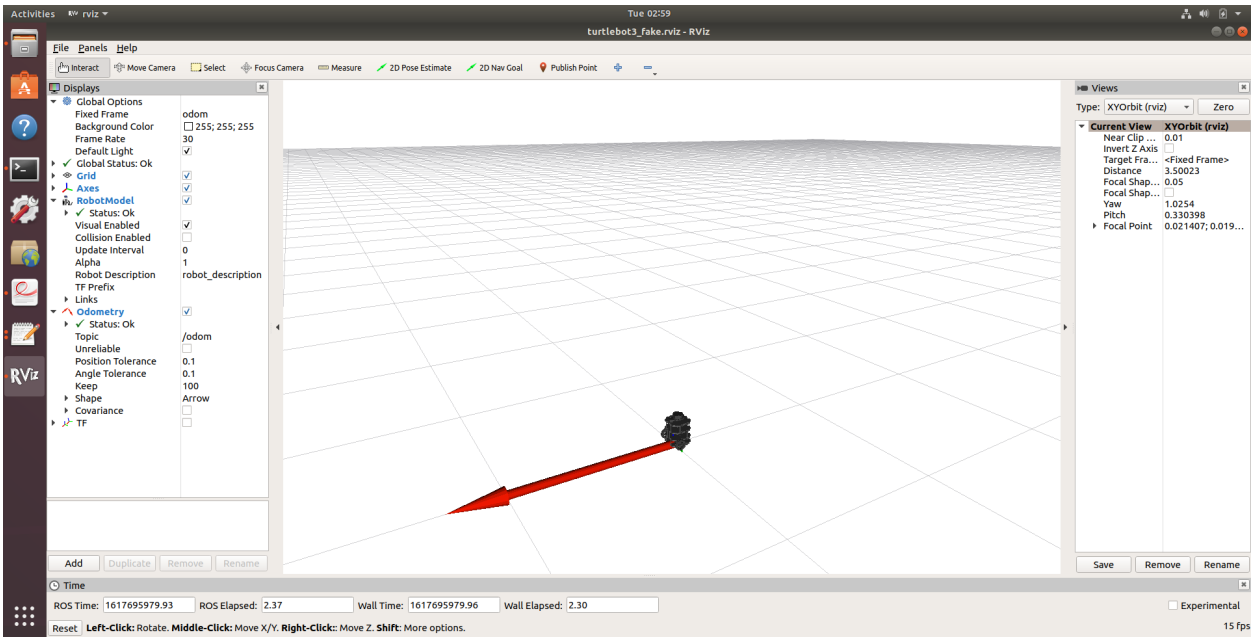
The TurtleBot 3 RVIZ simulation is created and controlled by the `turtlebot3_fake_node` node. This node generates the model of the TurtleBot in RVIZ and allows for it to be controlled using the Teleop node.

To run the TurtleBot 3 simulation first run the following command.

```
roslaunch turtlebot3_fake turtlebot3_fake.launch
```

This will open Turtlebot3 model in RVIZ. This model is cretaed using URDF definitions as we did in week 3 tutorial for our robotic arm.

Three nodes are started: robot_state_publisher, rviz, and turtlebot3_fake_ node. Then you should see the following screen.



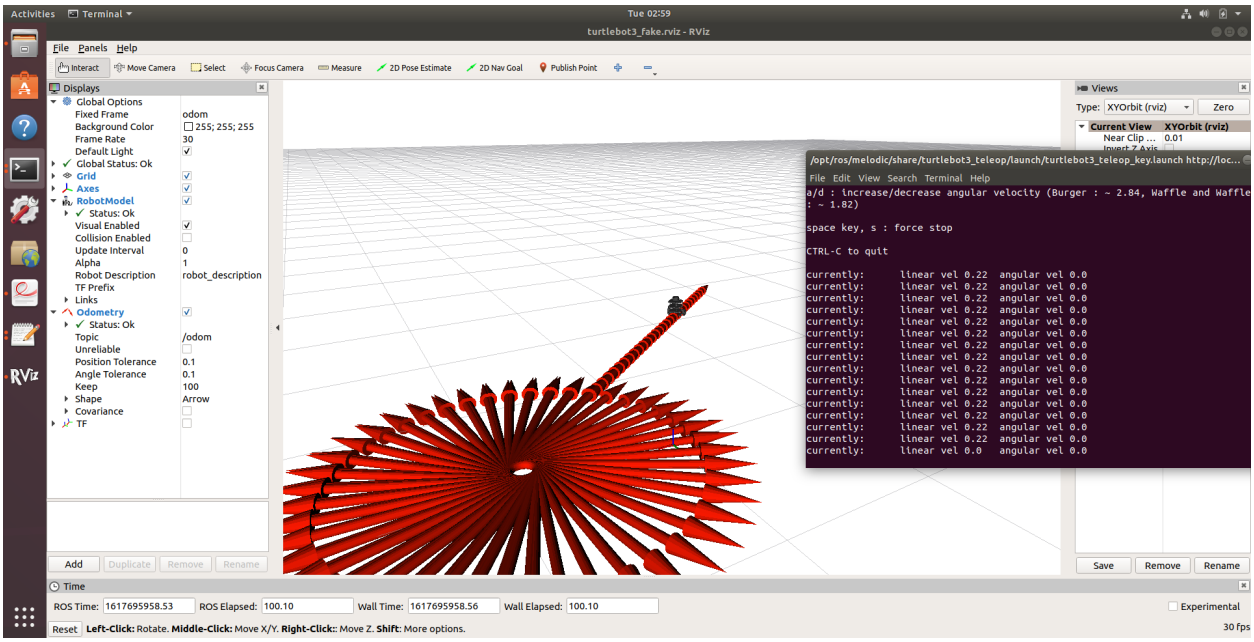
Now let's control the robot using the keyboard. Open a new terminal and run,

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

You can control the robot by using the following keys

- w/x : increase/decrease linear velocity
- a/d : increase/decrease angular velocity
- space key, s : force stop

look at the /odem topic and observe it in RVIZ. This the odometry of the robot.



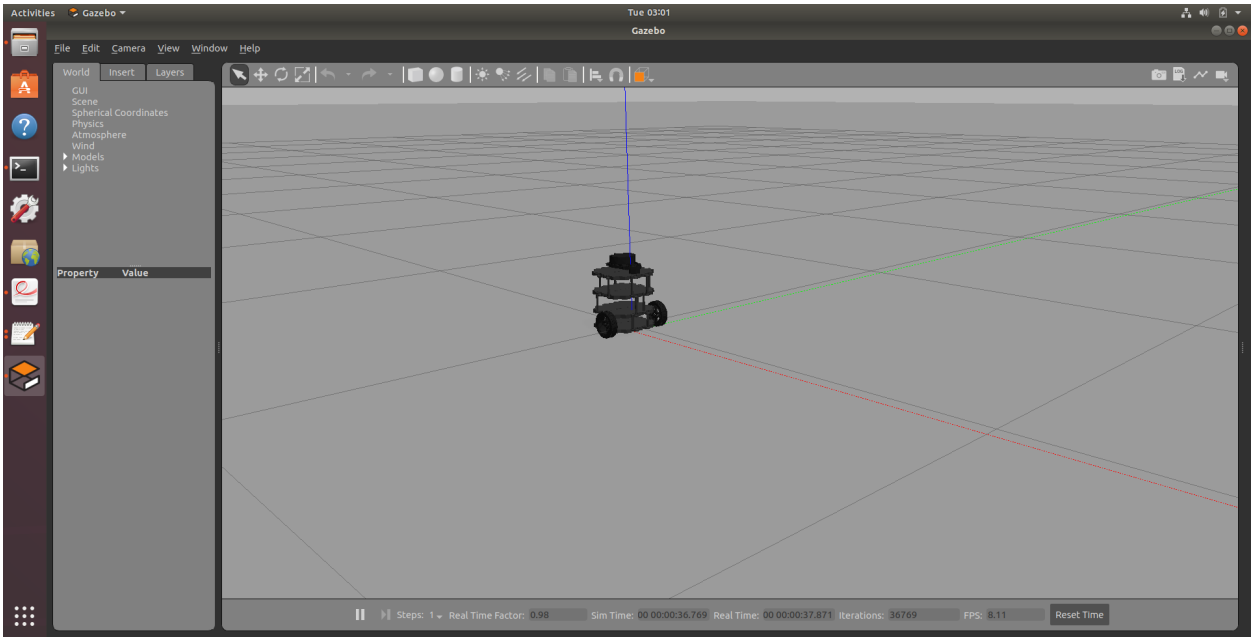
Next, we will try TurtleBot 3 in the Gazebo 3D simulator. Close all terminal windows before proceeding to the next section

13.1.2 Launching TurtleBot 3 simulation in Gazebo

Now let's run TurtleBot3 in Gazebo. Run the following command

```
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

This will open a Gazebo simulation using Tutrlebot 3 as shown in the following figure. You can zoom in and out by pressing the right mouse button.



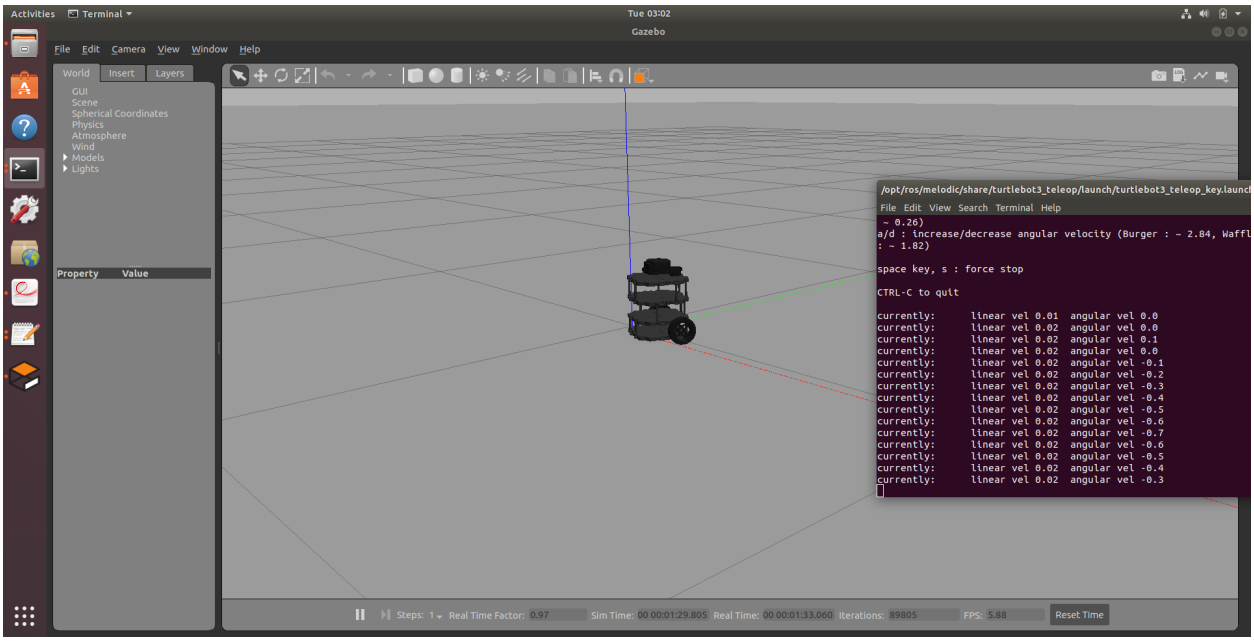
Now let's move the turtlebot around using the keybord.

Open a new terminal and run'

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

You can control the robot by using keys. You will see now the robot is moving in Gazebo.

- w/x : increase/decrease linear velocity
- a/d : increase/decrease angular velocity
- space key, s : force stop



Now let us try turtlebot on a challenging environemnt. Use Quit on the Gazebo application from the menu bar and press Ctrl + C in the terminal window to halt the process

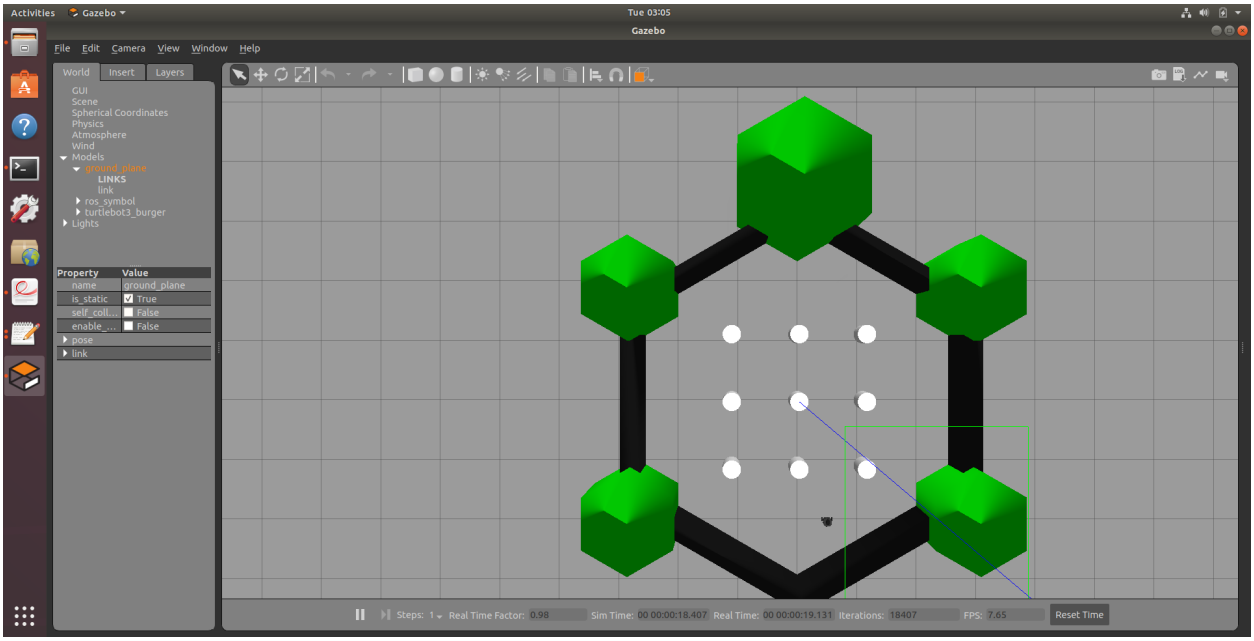
Now, in a new terminal type the following command to spawn the TurtleBot 3 model on the TurtleBot 3 world map:

```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

This command should produce a screenshot similar to the following:

You can see the Turlebot as a small black circle. Zoom in to (press right mouse button and drag) get a closer look.

To move the robot using the keyboard, launch the `turtlebot3_teleop_key.launch` file as explained earlier.



13.1.3 Turtlebot Autonomous Operation

Now let's move the Turtlebot Autonomously.To view this application, hit Ctrl + C in the terminal window running the keyboard teleop process, and close the window. Open a new terminal window run the the following commands

```
roslaunch turtlebot3_gazebo turtlebot3_simulation.launch
```

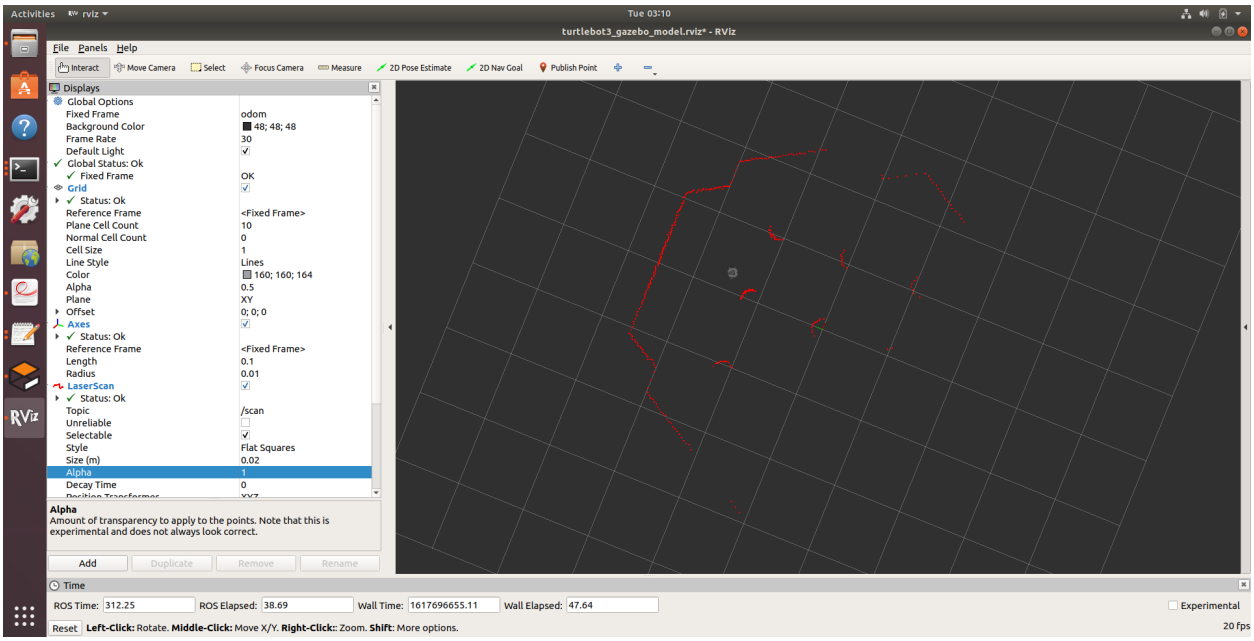
This will run the turtlebot in the maze autonomously using laser scanner data.

Now let's see how the laser scanner data looks like in Rviz. The laser scanner sends laser beams around the sensor and measures the distance to nearby objects. You can see the measurements from each laser beam as a small point marker in Rviz. To see laser data run the following command.

```
roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
```

This will show the laser data. You need to add the laser_scan module in Rviz display and configure the `/scan` topic as the input. This is already done.

When the robot moves around you would see that the laser data is changing.



13.1.4 Moving Turtlebot Autonomoulsy Using Python

In this section let's learn how we can move Turtlebot around in the Gazebo simulation environment using laser scan data. We need to publish velocity commands to `/cmd_vel` to move the robot. To read the laser scan data the following python code subscribe to the `/scan` topic. It is outputting each laser beam's distance to the nearby objects in an array. There are 360 laser beams around the sensor, and therefore there are 360 elements in the array. 0 th element of the array corresponds to the first laser beam located at the front.The following code read the laser scan data from the front, left and right laser beams and try to avoid the obstacle if they are within 0.5 m.

```
cattkin_create_pkg turtlebot3_move rospy sensor_msgs geometry_msgs
```

Then inside the `src` folder create a file called `autonomous_turtlebot.py` and copy the following code.

```
#!/usr/bin/env python
import rospy # Python library for ROS
from sensor_msgs.msg import LaserScan # LaserScan type message is defined in sensor_
↪msgs
from geometry_msgs.msg import Twist # Twist messages for velocity commands



def callback(dt):
   print '-----------------------------------------'
   print 'Range data at 0 deg:' ,dt.ranges[0]
   print 'Range data at 15 deg:', dt.ranges[15]
   print 'Range data at 345 deg:' ,dt.ranges[345]
   print '-----------------------------------------'

   thr1 = 0.5 # Laser scan range threshold
   thr2 = 0.5
   if dt.ranges[0]>thr1 and dt.ranges[15]>thr2 and dt.ranges[345]>thr2: # Checks if␣
↪there are obstacles in front and
                                                                        # 15 degrees␣
↪left and right (Try changing the
                                                                                    ␣
↪                                    # the angle values as well as the thresholds)
       move.linear.x = 0.2 # go forward (linear velocity)
       move.angular.z = 0.0 # do not rotate (angular velocity)
   else:

       move.linear.x = 0.0 # stop
       move.angular.z = 0.5 # rotate counter-clockwise
       #if dt.ranges[0]>thr1 and dt.ranges[15]>thr2 and dt.ranges[345]>thr2:
           #move.linear.x = 0.5
           #move.angular.z = 0.0
   pub.publish(move) # publish the move object


# Let's create the node publishers and subscribers

rospy.init_node('obstacle_avoidance_node') # Initializes a node

pub = rospy.Publisher("/cmd_vel", Twist, queue_size=10)
# Publisher object which will publish "Twist" type messages
# on the "/cmd_vel" Topic, "queue_size" is the size of the
# outgoing message queue used for asynchronous publishing




sub = rospy.Subscriber("/scan", LaserScan, callback)
# Subscriber object which will listen "LaserScan" type messages
# from the "/scan" Topic and call the "callback" function
# each time it reads something from the Topic


move = Twist() # Creates a Twist message type object to publish velocity commands to␣
↪the robot
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':


   try:
       rospy.spin() # Loops infinitely until someone stops the program execution
   except rospy.ROSInterruptException: pass
```

Try to understand what is happening in the code. Then make it an executable.

Then run the Gazebo simulator by

```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Now run the program that you wrote in a new terminal

```
rosrun turtlebot3_move autonomous_turtlebot3.py
```

You would see that the robot is moving autonomously in the maze.

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search