Introduction to robot operating system

Adaptive robotics minor - fontys university of applied science

Strik, Leonardo - Ayoub, Hussam

Introduction:

This tutorial series was made as part of a project that aims to make a set of tutorials for the simulation of robots in **r**obot **o**perating **s**ystem (ROS), that can be utilised through a cloud-based computing platform. It will allow students or robotics enthusiasts around the world with limited hardware to run their simulations in the cloud, and start their journey in the Robotics field.

The first part of these tutorials consists of three introductions. These explain basic commands and functionality of the software that will be used in the rest of the tutorials. The first is on Ubuntu, the operating system that ROS runs on. The second is on ROS itself. The third introduction is into Gazebo, the simulation engine that is used in conjunction with ROS.

After these introductions, the tutorials themselves are up.

The first tutorials describe how to make and set up a robot model in a **u**nified **r**obot **d**escription **f**ormat (URDF). This file can be used to simulate the robot and its sensors in Gazebo.

Then mapping and navigation are described, as well as how to use ROS control to control the robot. After this, how to control the manipulator by using ROS Industrial and ROS MoveIt!.

Finally, a way to integrate MATLAB and MATLAB Simulink models in ROS through code generation is shown.

To follow these tutorials, there is a package you can download to work along.

These tutorials are made to be used with ROS Melodic Morenia and Ubuntu 18.04.4 Bionic Beaver.

Table of contents

1 . Introduction to Ubuntu:

Ubuntu is a distribution (*distro*) of Linux, an open source Operating System (OS). It is the OS that ROS runs on, and therefore the OS we will be using in these tutorials. Usually, software in Ubuntu is distributed in so-called *packages.* These packages are stored in *repositories.* These packages can be downloaded, installed and managed using the **a**dvanced **p**ackage **t**ool (apt).
Like using apt, many tasks in Ubuntu can or must be performed via the terminal interface. For a proper use of Ubuntu and ROS, it is essential to be familiar with the terminal interface, and know at least the basics of how to use it. Below are some terminal commands and keyboard shortcuts that will prove useful during the tutorials described. If you want to learn more about Ubuntu or the terminal interface, we suggest looking at this tutorial.

Basic terminal commands :

sudo: Short for **s**uper**u**ser **do**. Written before other commands, gives the command you are executing privileges to access certain folders on your computer.

```
$ sudo <command> # runs the specified command with superuser privileges
```

apt : Short for **a**dvanced **p**ackage **t**ool. Used to manage Ubuntu packages, like installing, updating or removing them. Apt usually needs to be called with superuser privileges.

```
$ sudo apt install <package_name>  # installs a package called <package_name>
$ sudo apt remove <package_name>  # removes a package called <package_name>
```

ls : Short for **l**i**s**t. Lists files and folders in the current working directory.

```
$ ls # lists the contents of the current directory
$ ls <path> # lists the contents of the directory at the specified path
```

cd: Short for **c**hange **d**irectory. Command that is used to navigate to other directories than the current working directory.

```
$ cd <path> # changes the working directory to the one specified
$ cd # changes the working directory to the home directory
$ cd .. # changes the working directory to the directory above the current one
```

mkdir: Short for **m**a**k**e **dir**ectory. Used to create a new directory.

```
$ mkdir one # makes a new directory in the current working directory named one
```

cp: Short for **c**o**p**y. Can be used to copy files or directories.

```
$ cp one.txt two.txt # copies one.txt into a file called two.txt
$ cp one.txt one/ # copies one.txt into a directory named one
$ cp -R one/ two/ # copies a directory named one into a directory named two
```

mv: Short for **m**o**v**e. Moves files or a directory.

```
$ mv one.txt two.txt # moves one.txt into a file called two.txt
$ mv one.txt one # moves one.txt into a directory named one
$ mv  one two # moves a directory named one into a directory named two
```

rm : Short for **rm**ove. Used to remove a file or directory.

```
$ rm one.txt # removes one.txt
$ rm -r one # recursively removes a directory named one
```

man: Short for **man**ual. Displays the manual page for a command.

```
$ man <command> # displays the manual page for the specified command
```

And, two simple commands, that don't require extra syntax, to help you navigate the terminal interface:
pwd: Short for **p**rint **w**orking **d**irectory. prints current directory.

history: Prints your command history. Useful if you want to know a command you typed some time ago.

Basic keyboard shortcuts:

Note: these keyboard shortcuts may or may not work on non-default Ubuntu installations.
Note: When in a terminal window, normal shortcuts like Ctrl + t (new tab), Ctrl + c (copy), and Ctrl + v (paste) need to be accompanied by a Shift. So Ctrl + Shift + t, Ctrl + Shift + c, and Ctrl + Shift + v, respectively.

Ctrl + Alt + t: Opens a new terminal window.-
Ctrl + c: Kills the process currently running in this window.
Tab: Auto completes the command that's currently being typed. Only works if there's one option.
Tab (twice): Shows all possible options to complete the command currently being typed.
Lastly, it's useful to know that using the *up* and *down* arrow keys, you can navigate the history of input commands. So up arrow to see the last input command, and up again to go back one, and so on.

2 . Introduction to ROS:

ROS, or Robot Operating System, is a collection of tools and software libraries we will be using to simulate and control our robot.
Broadly, ROS works by running processes, called *nodes*. These nodes are each responsible for a separate task, like path planning or driving the robot motor. These nodes communicate by sending data to, and reading data from, channels, called *topics.* Sending data to these topics is called *publishing*, and reading from them is referred to as *subscribing.* These topics, and which node is publishing and subscribing to which topic, are managed by the *ROS master.* ROS software is, usually, organized in *packages.* Throughout this tutorial, we will be using and installing several different ROS packages. These ROS packages contain the nodes they need to work, along with so-called *launch files*. These launch files tell ROS what nodes to launch in order to perform specific tasks. In the tutorials, we will build our own packages, nodes and launch files.

Installing and configuring the ROS environment:

Please visit the ROS installation tutorial for installation instructions.

ROS and the command line:

To be able to use ROS effectively, a basic knowledge of using the command line to navigate and interact with ROS is essential. Below, a rundown of the commands used in the tutorial is provided. As with Ubuntu, we encourage readers who want to learn more to check out the ROS tutorials for more information.

Basic ROS commands:

roscore: Starts op the ROS master.
rosrun: Runs an executable from a ROS package directly.

```
$ rosrun <package_name> executable.py # runs executable.py from the specified package
```

roslaunch: Launches a package from a specified launch file.

```
$ roslaunch <package_name> <launch_file>.launch # launches a package from the specified launch file
```

roscd: Allows to change the directory by using a package name.

```
$ roscd package # changes the working directory to the one the package is in
```

rostopic: A command line tool to display information about topics.

```
$ rostopic list  # lists all active topics
$ rostopic echo  # prints topic messages to screen
$ rostopic info  # prints information about the active topics
```

rosnode: A command line tool to display information about nodes.

```
$ rosnode info   # prints information about node
$ rosnode kill   # kills a running node
$ rosnode list   # lists all active nodes
```

catkin: Catkin is the build environment for ROS. It is used to create and build ROS packages.

```
$ catkin_create_pkg # make a new package
$ catkin_make # build the catkin workspace
```

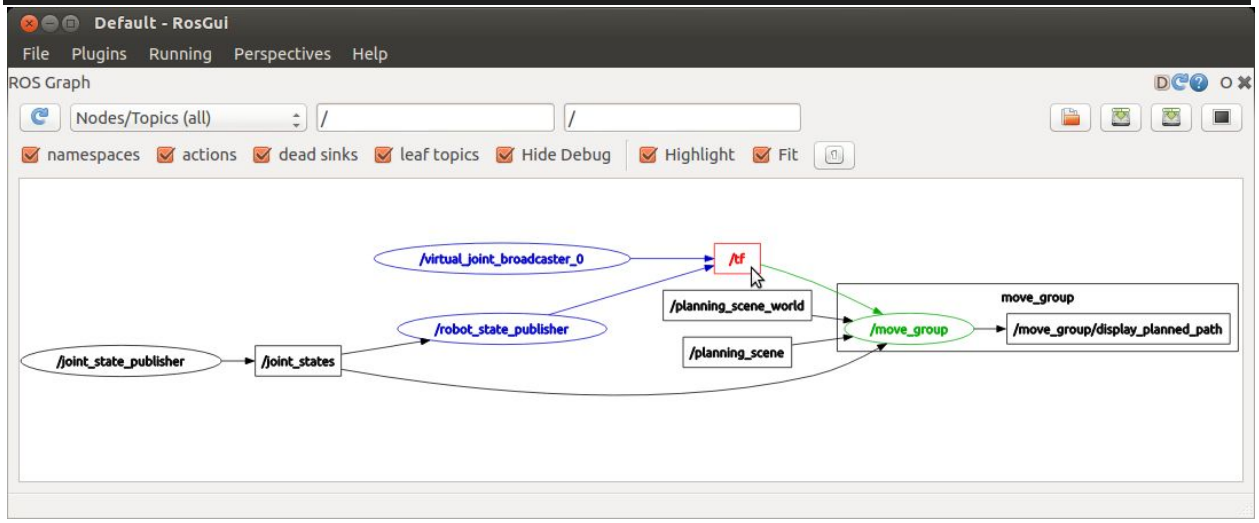
ROSQt toolkit

ROSQt, or RQt, is a Qt-based toolkit that helps us visualise, and change, multiple aspects of our robot and ROS programming. Qt is a framework, used to create the **g**raphical **u**ser **i**nterface (GUI) for this toolkit. It is beyond the scope of these tutorials.

Below, we will take a look at the three main RQt tools that might come in handy during these tutorials, rqt_graph, rqt_tf_tree, and rqt_reconfigure. For a broader overview of RQt tools, please visit this link.

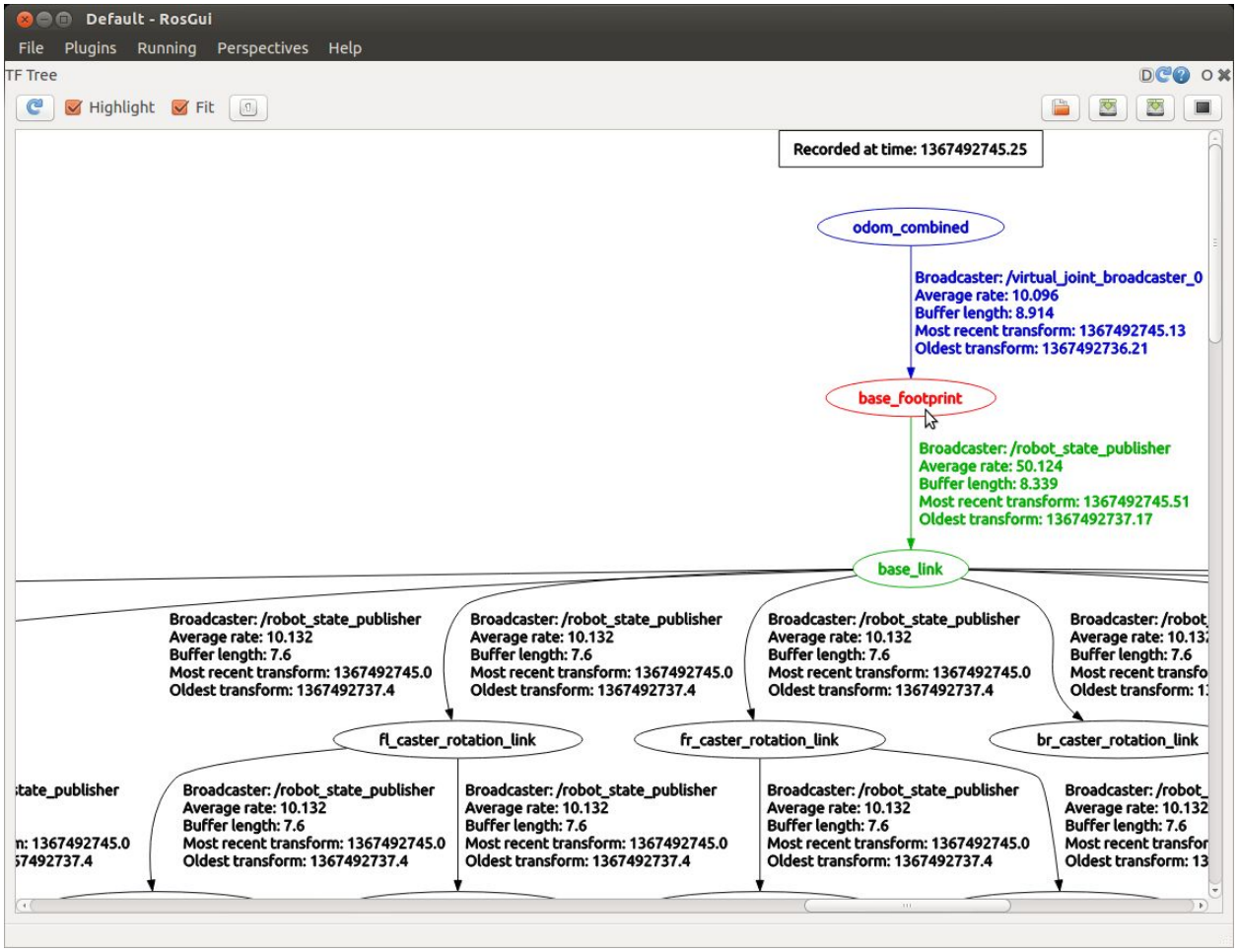
rqt_graph: The RQt graph plugin allows us to visualise all active ROS nodes and topics, and the flow of data between them, as can be seen in figure 1.

```
$ rosrun rqt_graph rqt_graph # Runs the RQt graph plugin
```



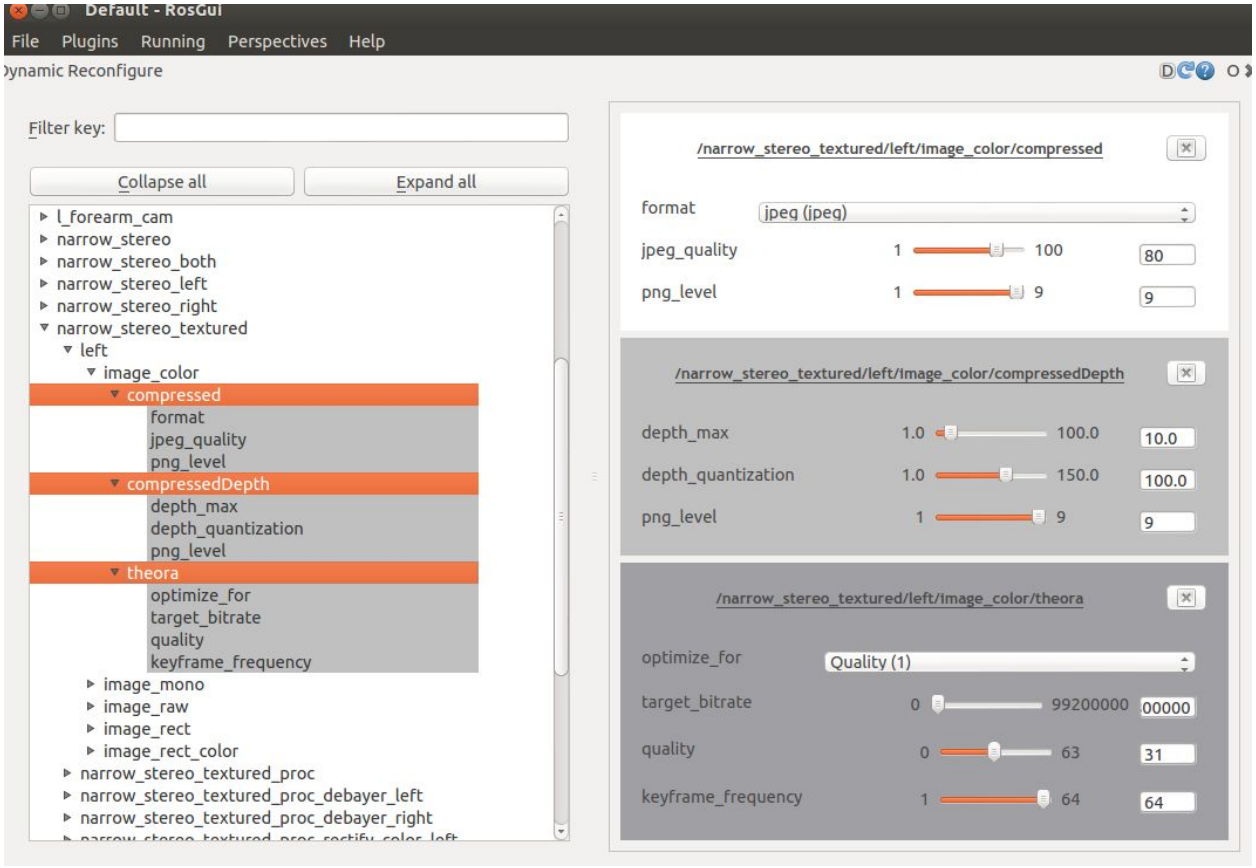
rqt_tf_tree: The RQt tf_tree plugin allows us to visualise the ROS TF frame tree. This is a representation of your robot where the relation coordinate frames of the robot links are shown. It also gives an overview of the general structure of the robot.

```
$ rosrun rqt_tf_tree rqt_tf_tree # Runs the RQt tf_tree plugin
```



rqt_reconfigure: Usually, ROS nodes have certain parameters that are set from a configuration file when they are launched. These parameters change robot behaviour, either by changing certain values, or enabling or disabling certain behaviours entirely. The RQt reconfigure plugin allows us to change parameters of ROS nodes while they are running. This is incredibly useful when trying to tune the parameters on your navigation, for example, without having to relaunch it.

```
$ rosrun rqt_reconfigure rqt_reconfigure # Runs the RQt reconfigure plugin
```



3 . Introduction to Gazebo:

Gazebo is a 3D robot simulation tool. It is similar to other simulation suites such as Unity, but includes high fidelity physics simulation, and easily integratable sensors and interfaces. This makes it very well-suited to be used as a simulator with ROS.

In Gazebo, robot models are defined in the Unified Robot Description Format (URDF). This is an XML format in which robots can be described.

In these tutorials, we will be using Gazebo to simulate our robot and its environment. Using Gazebo, we can also simulate the sensors on our robot, and the data they would produce. This allows us to test our mapping and navigation with simulated inputs. We will look at how to create our own robot description package with a URDF, how to load it into Gazebo, and how to use Gazebo plugins to simulate the sensors we want to use.

4 . Modeling a simple Robot:

In this tutorial we will take a look at how robot description packages are made. We will do this by building our own 2 link manipulator package, and launching it.
First of all we will start by writing a URDF file that contains the description of the robot links and joints, and later we will create a package to visualize the robot in rviz.
Before we start writing the package we should take a look at some concepts used in this tutorial:

Introduction:

4.1.1 URDF files

As mentioned before, URDF is an XML file format used in ROS to describe all elements of a robot. To use a URDF file in Gazebo, the URDF file must be converted to a Simulation Description Format (SDF), by adding some additional simulation-specific tags. This tutorial explains the necessary steps to successfully import your URDF-based robot in Gazebo, saving you from having to create a separate SDF file from scratch and duplicating description formats. Under the hood, Gazebo will then convert the URDF to SDF automatically.

4.1.2 Xacros:

XML-macros (Xacros) are used to simplify large URDF files. They can be used to split large URDF XML files into parts, define repeated variables in a central location, and more. This allows for easy changing of variables, and makes the URDF file more readable.

4.1.3 URDF and Solidworks:

Making a URDF manually, as shown in tutorial 4.2, can be a complicated and lengthy process form large or complicated robots. Luckily, there is a Solidworks plugin to export a Solidworks model as a URDF file. This plugin can be found on the ROS wiki. While designing the Solidworks model to be exported, some minor design considerations need to be taken into account. If you are interested in the process of exporting a URDF file from Solidworks, please follow this tutorial. Otherwise, a pre-made URDF file can be downloaded from SOMEWHERE.

4.2- Creating the robot package and simple robot control:

Now that you know the principles that will be applied in this tutorial, it's time to actually create the package. We will use catkin to create a package, write the URDF file for our robot, visualize it in rviz and control it through the joint state publisher.

4.2.1 Creating the package:

First we will use catkin to create a package for our robot called robot_description, and make two directories, one for the launch file and one for the URDF file.

Navigate to the source (src) directory in the catkin workspace (catkin_ws):

```
$ cd ~/catkin_ws/src
```

Create a package called robot_description with the following command:

```
$ catkin_create_pkg robot_description
```

Navigate to the directory that contains the robot_description package:

```
$ cd robot_description
```

This is the directory that will contain all necessary files for our package. In this directory, create two new directories named launch and urdf with the following command:

```
$ mkdir launch urdf
```

After creating these directories, navigate to the urdf directory:

```
$ cd urdf
```

In this directory we want to create the robot_description_urdf.urdf, using a text editor or an IDE. In this tutorial, we will be using Visual Studio Code (VSC) as an IDE. You can download and install VSC for free here.
Create the URDF file with the following command:

```
$ code robot_description_urdf.urdf
```

4.2.2 URDF file:

Now we made the package, and directories to contain our URDF and launch files, we will start writing the URDF file.
Our simple 2 link robot will consist of two links, connected by a joint. This will be described by four XML tags in our URDF file; a main robot tag, which contains the robot name, two link tags, one for each link, and a joint tag.

First, let's make the robot tag. We define the version of XML we're using, and make a robot tag for a robot named "my_robot":

```
<?xml version="1.0" ?>
<robot name="my_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">
```

First we will make a base link, named "base_footprint", which the other link will connect to. It will be a cylinder with length 1 and radius 0.1. The link has a pose, collision, and visual attribute. For now, we will set the pose to be 0, and the visual and collision to both be our cylinder:

```
<!-- First Link -->
   <link name="base_footprint">
       <pose>0 0 0 0 0 0</pose>
       <collision name="collision">
           <geometry>
               <cylinder length="1" radius="0.1" />
           </geometry>
       </collision>
       <visual>
           <origin rpy="0 0 0" xyz="0 0 0" />
           <geometry>
               <cylinder length="1" radius="0.1" />
           </geometry>
       </visual>
   </link>
```

Now we make the second link named “second_link”. This one also has a pose of 0, and a visual and collision mesh of our cylinder. The origin of this one, however, is offset by half the cylinder length.

```
<!-- Second Link -->
   <link name="second_link">

       <pose>0 0 0 0 0 0</pose>
       <collision name="collision">
           <geometry>
               <cylinder length="1" radius="0.1" />
           </geometry>
       </collision>
       <visual>
           <origin rpy="0 0 0" xyz="0 0 0.5" />
           <geometry>
               <cylinder length="1" radius="0.1" />
           </geometry>
       </visual>
   </link>
```

Thirdly, we will add our joint. We will set it to be of type “continuous”, which means it can rotate indefinitely. It will connect our two links.

```
   <!-- Joint -->
   <joint name="joint1" type="continuous">
       <origin xyz="0 0 0.5" rpy="0 0 0" />
       <parent link="base_footprint" />
       <child link="second_link" />
   </joint>
```

Don’t forget to close the robot tag at the end of your URDF!

```
</robot>
```

4.2.3 launch file:

Now that we made the URDF file, we want to be able to parse it into rviz or Gazebo, to actually see it. In order to do this, we'll need to write a launch file.

Navigate to the launch directory in the robot_description package:

```
$ cd ~/catkin_ws/src/robot_description/launch
```

Using VSC, create a launch file:

```
$ code robot.launch
```

This launch file will include some parameters, and all the ROS nodes we will need for now:

- robot_state_publisher
- joint_state_publisher
- spawn_model
- rviz

First, we will give the launch file some parameters, to be able to find our URDF file.

```
<launch>
   <param name="robot_description" command="$(find xacro)/xacro '$(find
robot_description)/urdf/robot_description_urdf.urdf'" />
```

Next, we'll give the launch file a list of the nodes we want it to start for us; both state publishers, the URDF model spawner, and rviz:

```
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" />
      <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
   <param name="use_gui" value="true" />
 </node>
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false"
output="screen" args="-urdf -x 0 -y 0 -z 0 -model robotcleaner -param
robot_description" />
 <node name="rviz" pkg="rviz" type="rviz" />
</launch>
```

After saving this file, we will need to build our catkin workspace. To do this, navigate to the catkin workspace, and use the catkin_make command:

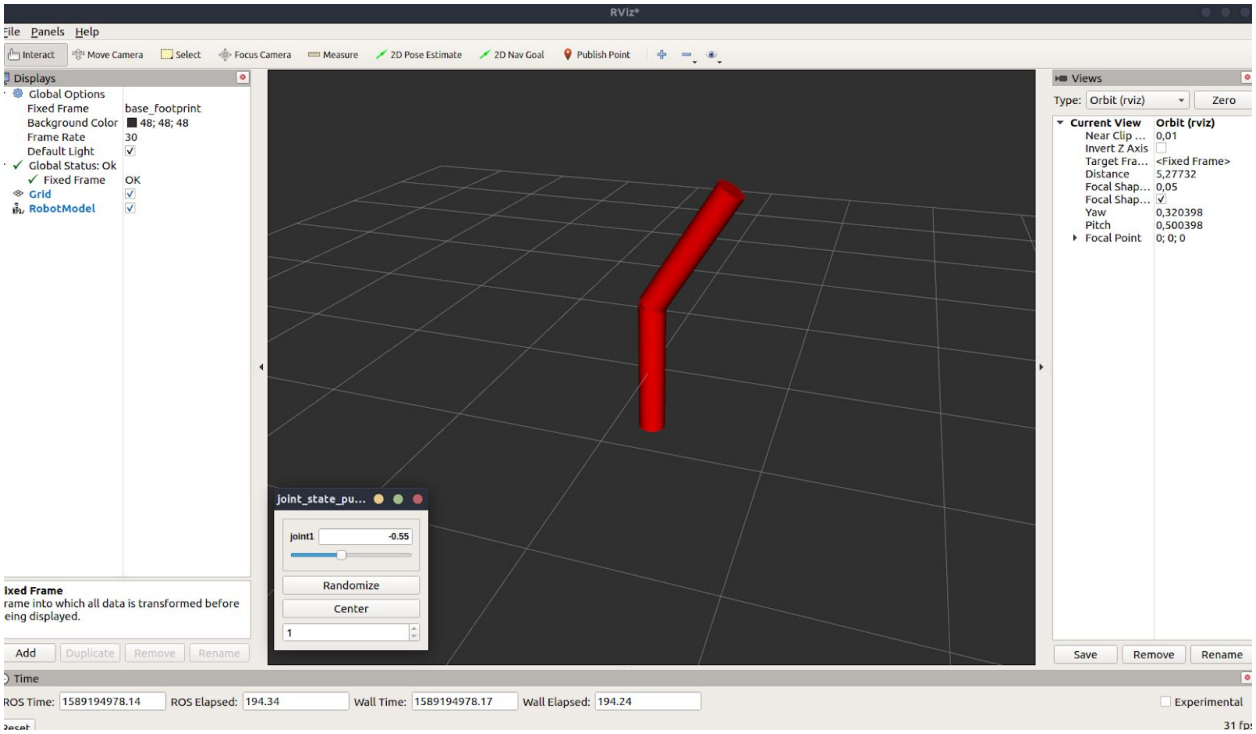
```
$ cd ~/catkin_ws
$ catkin_make
```

Now let's launch the package :

```
$ roslaunch robot_description robot.launch
```

All the nodes specified in the launch file will be started. Most nodes run in the background, but rviz and the joint state publisher will show a GUI.The nodes specified in the launch file will start in the background, rviz and joint_state_publisher will start in gui.

In rviz, we want to change the Fixed Frame in our displays panel to the base_footprint. Now click add in the bottom left, and add the robot model. We can now see our robot model, and change the joint state by using the joint state publisher GUI.



Congratulations, you have built your first robot description package!
All the files made in this tutorial can be found in their complete form in github.

5 . Heterogeneous robot:

5.1 intro:

In this chapter we will create a description package of a heterogeneous robot. The robot consists of two main parts: a holonomic platform with four wheels and an arm with six degrees of freedom. This is the robot that will be used for the rest of the tutorials.

Note: It is not recommended to follow this tutorial before finishing the previous tutorials.

5.3 Gazebo plugins:

Now that we are familiar with the generated package from solidworks we can start adding some Gazebo plugins to the URDF file.

Before adding plugins let's take a look at what Gazebo plugins actually are.
According to gazebosim.org a gazebo plugin is a chunk of code that is compiled as a shared library and inserted into the simulation. The plugin has direct access to all the functionality of Gazebo through the standard C++ classes.

Plugins are useful because they:

- let developers control almost any aspect of Gazebo
- are self-contained routines that are easily shared
- can be inserted and removed from a running system

There are several types of gazebo plugins:

World, Model, Sensor, System, Visual, and GUI

5.3.1 Ros planner Movement:

Before starting with the planner movement plugin we will prepare the workspace and the URDF file to be used with xacros. This will simplify the modeling.

First navigate to urdf folder in the example_urdf package exported from solidworks:

```
$ cd ~/catkin_ws/src/example_urdf/urdf
```

Rename the example_urdf.urdf file to example_urdf.urdf.xacro:

```
$  mv example_urdf.urdf example_urdf.urdf.xacro
```

Create a file called example_gazebo.xacro:

```
$ code example.gazebo.xacro
```

Now let's edit the URDF file:
After the main robot tag in the beginning of the file we will do the following:

- Include the gazebo.xacro file
- add an extra virtual link called base_footprint
- add a joint to attach the whole robot model to the link
- Create code regions

```
<robot name="example_urdf" xmlns:xacro="example_robot">
 <xacro:include filename="$(find example_urdf)/urdf/example.gazebo.xacro" />

 <!-- #region footprint -->
 <link name="base_footprint" />
 <joint name="base_joint" type="fixed">
   <parent link="base_footprint" />
   <child link="base_link" />
   <origin xyz="0 0 0" rpy="0 0 0" />
 </joint>
 <!-- #endregion footprint-->
```

Now that we edited the URDF file, and changed its name, we should amend the launch file to reflect these changes.

```
<launch>
 <include file="$(find gazebo_ros)/launch/empty_world.launch" />
   <param    name="robot_description"    command="$(find    xacro)/xacro    '$(find
example_urdf)/urdf/example_urdf.urdf.xacro'" />
       <node       name="joint_state_publisher"       pkg="joint_state_publisher"
type="joint_state_publisher" />
       <node       name="robot_state_publisher"       pkg="robot_state_publisher"
type="robot_state_publisher" />
  <node  name="urdf_spawner"  pkg="gazebo_ros"  type="spawn_model"  respawn="false"
output="screen"  args="-urdf  -x  0  -y  0  -z  0    -model  example_urdf  -param
robot_description" />
</launch>
```

Now we are ready to start using our first gazebo plugin!
We want to use the object controller plugin to take commands from the cmd_vel topic, and control our robot based on the virtual base_footprint link.

To do this, add the following in the example.gazebo.xacro:

```
<?xml version="1.0" encoding="utf-8"?>

<robot name="example_urdf" xmlns:xacro="example_robot">
 <!--#region Holonomic-->
 <gazebo>
 <plugin name="object_controller" filename="libgazebo_ros_planar_move.so">
     <commandTopic>cmd_vel</commandTopic>
     <odometryTopic>odom</odometryTopic>
     <odometryFrame>odom</odometryFrame>
     <odometryRate>20.0</odometryRate>
     <robotBaseFrame>base_footprint</robotBaseFrame>
   </plugin>
 </gazebo>
<!--#endregion holonomic-->
```

To test the planar plugin, let's install the teleop_twist_keyboard package:

```
$ sudo apt install ros-melodic-teleop-twist-keyboard
```

Now let's launch the example_urdf package:

```
$ roslaunch example_urdf gazebo.launch
```

Run the teleop keyboard:

```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

You can control the robot movement through the teleop node, for holonomic movement hold shift.
Now that we've added the planar plugin, let's give our robot some colour!

5.3.2 Materials:

Adding a material is a pretty straightforward task in Gazebo. It's done by referencing the link by its name, and defining a material from the list of predefined materials (found here).

In the example.gazebo.xacro, create a material region, and add a colour for the base_link. Feel free to pick your own colour from the linked list.

```
<gazebo reference="base_link">
   <material>Gazebo/Red</material>
 </gazebo>
```

5.3.3 Virtual sensors & data synthesis:

Now we will be looking at plugins that allow us to simulate sensors, and synthesize data to test our robot. Firstly, we will look at the ROS laser plugin, which allows us to simulate a LIght Detection And Ranging (LIDAR) sensor. Then we will take a look at the ROS Kinect plugin, which allows us to simulate a camera sensor..

5.3.3.1 ROS laser

In this section we will add the libgazebo_ros_laser.so plugin to our example.gazebo.xacro file in order to simulate two LIDARs.
We will have to make a physical model for our LIDARs, and add the Gazebo plugin that simulates them afterwards.

First, let's make the physical model for the LIDARs, in our example_urdf.urdf.xacro. We'll make two cylinders with a length of 0.0315, and a radius of 0.02. Let's put these links in their own region:

```
<!-- #region laser links-->
<link name="laser1">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>robot
      <cylinder length="0.0315" radius="0.02" />
    </geometry>
  </visual>
</link>
<link name="laser2">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <cylinder length="0.0315" radius="0.02" />
    </geometry>
```

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

Now let's join our two links to our base link:

```
 <joint name="laser1_joint" type="fixed">
  <origin xyz="0.1 -0.0 0.01" rpy="0 0 0" />
  <parent link="base_link" />
  <child link="laser1" />
</joint>
<joint name="laser2_joint" type="fixed">
  <origin xyz="-0.1 0.0 0.01" rpy="0 0 3.14" />
  <parent link="base_link" />
  <child link="laser2" />
</joint>
<!--#endregion links-->
```

Now that we have the two models, we can add the laser plugin. To do so, add the following in the example.gazebo.xacro file:

```
<!--#region laser-->
 <gazebo reference="laser2">
   <sensor type="ray" name="fullradius">
     <pose>0 0 0 0 0 0</pose>
     <visualize>true</visualize>
     <update_rate>40</update_rate>
     <ray>
       <scan>
         <horizontal>
           <samples>720</samples>
           <resolution>1</resolution>
           <min_angle>-1.770796</min_angle>
           <max_angle>1.770796</max_angle>
         </horizontal>
       </scan>
       <range>
         <min>0.10</min>
         <max>3.0</max>
         <resolution>0.01</resolution>
```

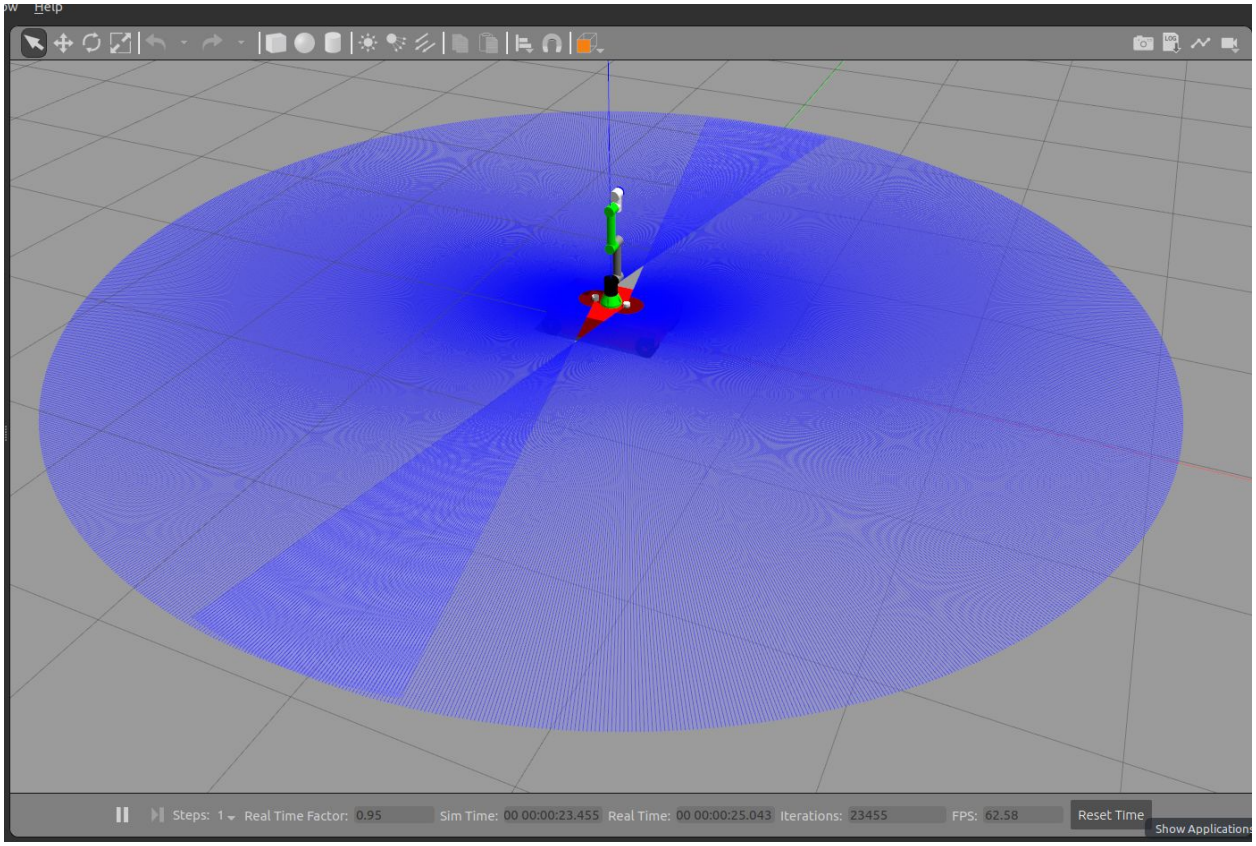
```
      </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
                          <plugin      name="gazebo_ros_head_hokuyo_controller"
filename="libgazebo_ros_laser.so">
    <topicName>scan2</topicName>
      <frameName>laser2</frameName>
    </plugin>
  </sensor>
 </gazebo>

 <gazebo reference="laser1">
  <sensor type="ray" name="halfradius">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>40</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>720</samples>
          <resolution>1</resolution>
          <min_angle>-1.770796</min_angle>
          <max_angle>1.770796</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.10</min>
        <max>3.0</max>
        <resolution>0.01</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
```

```
        <stddev>0.01</stddev>
      </noise>
    </ray>
                          <plugin      name="gazebo_ros_head_hokuyo_controller"
filename="libgazebo_ros_laser.so">
    <topicName>scan1</topicName>
      <frameName>laser1</frameName>
    </plugin>
  </sensor>
 </gazebo>
<!--#endregion laser-->
```

Notice that we can edit the plugin values to match what we are trying to achieve. For example, we can adjust the laser angle in:

```
          <min_angle>-1.770796</min_angle>
          <max_angle>1.770796</max_angle>
```

We can also change the `<visualize>true</visualize>` value to visualize the laser rays in gazebo.
For more information about the plugin please refer to gazebo :tutorial page.
Below is a screenshot from gazebo to the current state of the robot.



5.3.3.2 ROS Kinect

The final plugin we are adding to our robot is libgazebo_ros_kinect.so which will simulate a camera sensor. We will start by using xacros to define some global variables in our example_urdf.urdf.xacro to be used later in the link origin:

```
<xacro:property name="r200_cam_rgb_px" value="0.380" />
<xacro:property name="r200_cam_rgb_py" value="-0.07" />
<xacro:property name="r200_cam_rgb_pz" value="-0.031" />
<xacro:property name="r200_cam_depth_offset" value="0.03" />
```

Now we will add a link to serve as our camera body in our example_urdf.urdf.xacro, and join it to the base link, just as we did for the LIDARs. We will also use joints to define the positions of the different camera frames, to be used later:

```
<!--#region cam links-->
<joint name="camera_joint" type="fixed">
  <origin xyz="${r200_cam_rgb_px} ${r200_cam_rgb_py} ${r200_cam_rgb_pz}" rpy="0
0 0" />
  <parent link="base_link" />
```

```
  <child link="camera_link" />
 </joint>

 <link name="camera_link">
   <visual>
     <origin xyz="0 0 0" rpy="1.57 0 1.57" />
     <geometry>
       <mesh filename="package://example_urdf/meshes/r200.dae" />
     </geometry>
   </visual>
   <collision>
     <origin xyz="0.003 0.065 0.007" rpy="0 0 0" />
     <geometry>
       <box size="0.012 0.132 0.020" />
     </geometry>
   </collision>
 </link>

 <joint name="camera_rgb_joint" type="fixed">
    <origin xyz="${r200_cam_rgb_px} ${r200_cam_rgb_py} ${r200_cam_rgb_pz}" rpy="0
0 0" />
   <parent link="camera_link" />
   <child link="camera_rgb_frame" />
 </joint>
 <link name="camera_rgb_frame" />

 <joint name="camera_rgb_optical_joint" type="fixed">
   <origin xyz="0 0 0" rpy="-1.57 0 -1.57" />
   <parent link="camera_rgb_frame" />
   <child link="camera_rgb_optical_frame" />
 </joint>
 <link name="camera_rgb_optical_frame" />

 <joint name="camera_depth_joint" type="fixed">
     <origin  xyz="${r200_cam_rgb_px}  ${r200_cam_rgb_py  +  r200_cam_depth_offset}
${r200_cam_rgb_pz}" rpy="0 0 0" />
   <parent link="camera_link" />
```

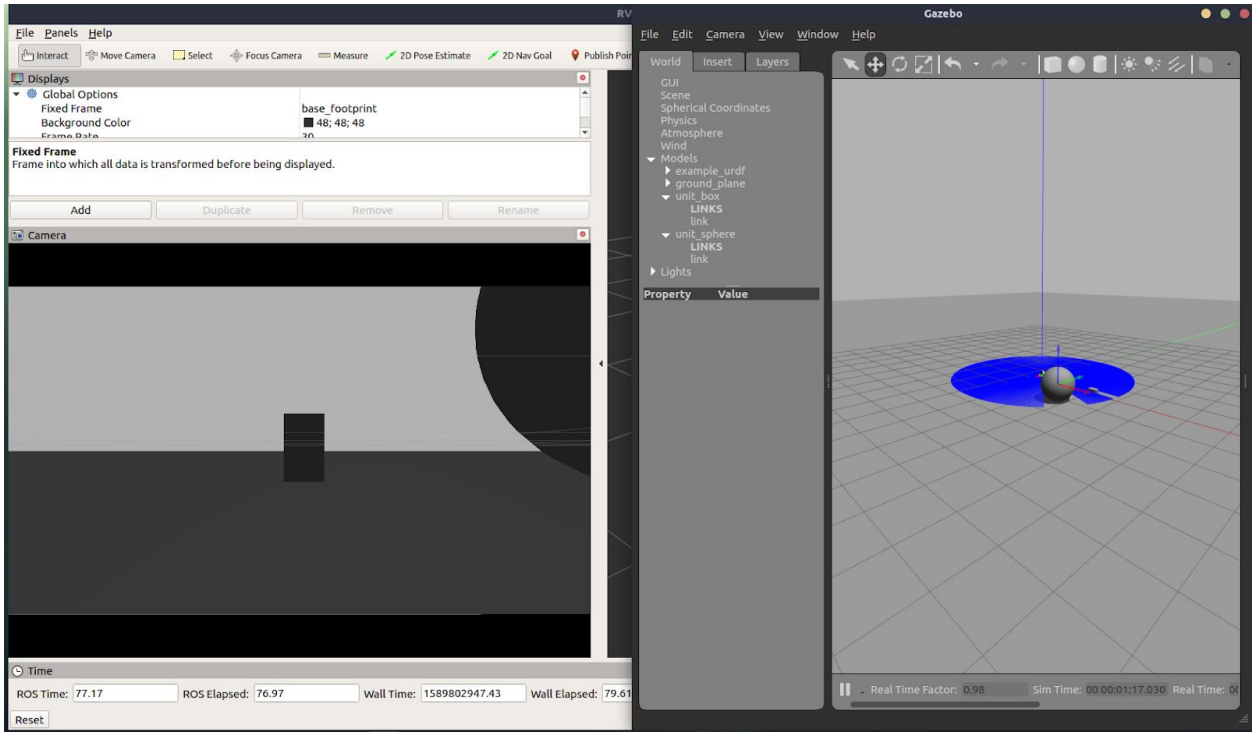
```
  <child link="camera_depth_frame" />
</joint>
<link name="camera_depth_frame" />

<joint name="camera_depth_optical_joint" type="fixed">
  <origin xyz="0 0 0" rpy="-1.57 0 -1.57" />
  <parent link="camera_depth_frame" />
  <child link="camera_depth_optical_frame" />
</joint>
<link name="camera_depth_optical_frame" />
<!--#endregion links-->
```

After defining the link and joints in our example_urdf.urdf.xacro, we can add the gazebo plugin that will simulate the camera in our example.gazebo.xacro by adding the following:

```
<!--#region cams-->
 <gazebo reference="camera_rgb_frame">
   <sensor type="depth" name="realsense_R200">
     <always_on>true</always_on>
     <visualize>true</visualize>
     <camera>
         <horizontal_fov>1.3439</horizontal_fov>
         <image>
             <width>1920</width>
             <height>1080</height>
             <format>R8G8B8</format>
         </image>
         <depth_camera></depth_camera>
         <clip>
             <near>0.03</near>
             <far>100</far>
         </clip>
     </camera>
     <plugin name="camera_controller" filename="libgazebo_ros_openni_kinect.so">
       <baseline>0.2</baseline>
       <alwaysOn>true</alwaysOn>
       <updateRate>30.0</updateRate>
       <cameraName>camera</cameraName>
```

```
      <frameName>camera_rgb_optical_frame</frameName>
      <imageTopicName>rgb/image_raw</imageTopicName>
      <depthImageTopicName>depth/image_raw</depthImageTopicName>
      <pointCloudTopicName>depth/points</pointCloudTopicName>
      <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>

<depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopicName>
      <pointCloudCutoff>0.4</pointCloudCutoff>
          <hackBaseline>0.07</hackBaseline>
          <distortionK1>0.0</distortionK1>
          <distortionK2>0.0</distortionK2>
          <distortionK3>0.0</distortionK3>
          <distortionT1>0.0</distortionT1>
          <distortionT2>0.0</distortionT2>
      <CxPrime>0.0</CxPrime>
      <Cx>0.0</Cx>
      <Cy>0.0</Cy>
      <focalLength>0</focalLength>
      <hackBaseline>0</hackBaseline>
    </plugin>
  </sensor>
 </gazebo>
<!--#endregion cams-->
```

Below is a visualization of the camera sensor in rviz.



And that concludes the modeling of your very own heterogenous robot! As before, all the files made in the tutorial can be found in their complete form on github.

5.4 Simulation environment:

Now that we've modeled our own robot, it's time to actually simulate it in Gazebo. To do so, we're going to clone a git repository to download a world file, and add it to our gazebo launch file.

The first step is to clone the git repository. To do this, you need to install git, if you don't already have it. To do so, open a terminal window and enter this command:

```
$ sudo apt install git
```

Now, let's navigate to our catkin workspace, and make a new directory called house_world to clone the repository into:

```
$ cd catkin_ws/src
$ mkdir house_world
```

Then, navigate to the directory you just made, initialize git, and clone the repository:

```
$ cd house_world
$ git init
```

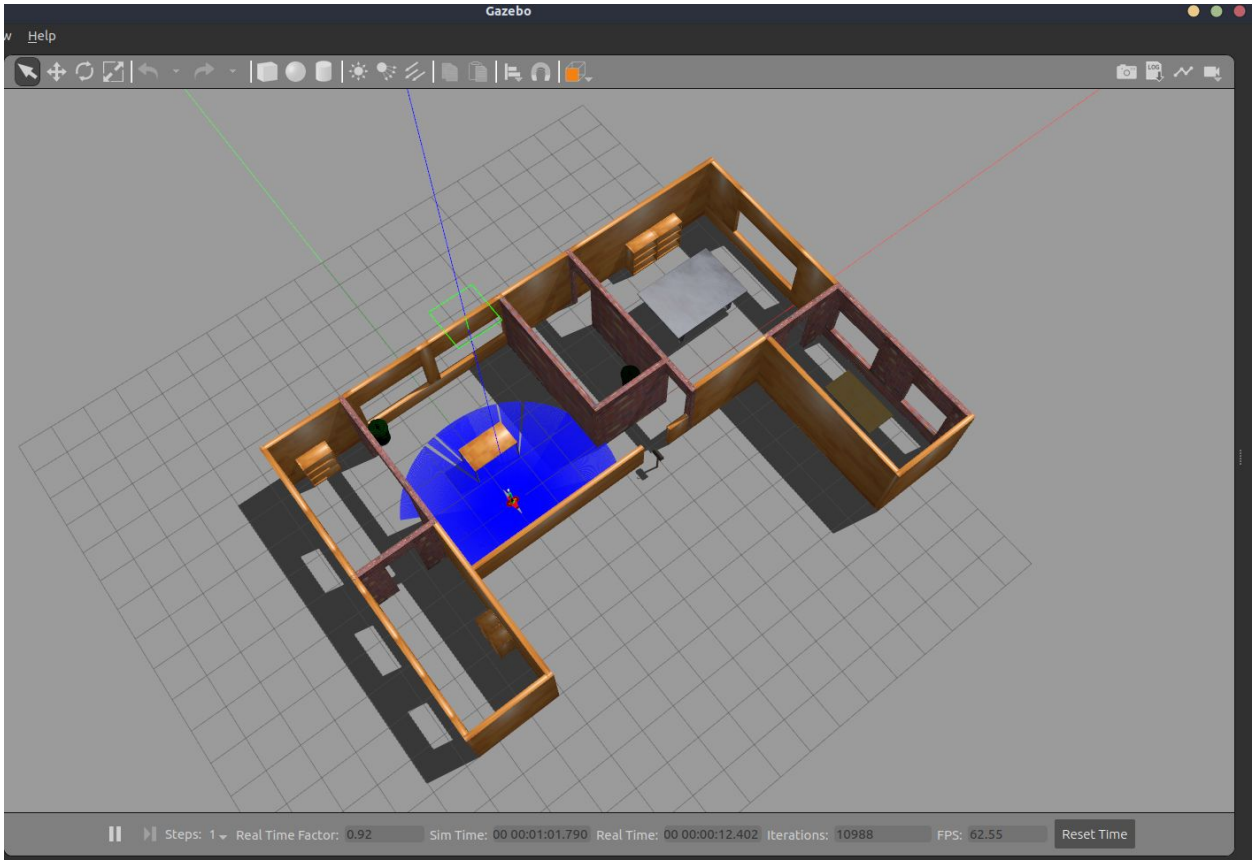
```
$ git clone
https://github.com/fontysrobotics/ARMinor-2020-Opensource-Virtual-Learning-Enviro
nment-AROVLE
```

If you look in the directory you just cloned into, you'll notice there are 2 new directories there, called worlds and models. These include all the models for a world with a house in it, which we will now add to the launch file.

Open gazebo.launch and add the following;

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
   <arg name="world_name" value="$(find example_urdf)/worlds/house.world"/>
   <arg name="paused" value="false"/>
   <arg name="use_sim_time" value="true"/>
   <arg name="gui" value="true"/>
   <arg name="headless" value="false"/>
   <arg name="debug" value="false"/>
 </include>
```

Now launch gazebo.launch to start the simulation.



6 . Mapping:

6.1 intro:

In this tutorial, we will look at how to use the robot we built in the previous tutorials to map an environment, like the house world we just made. We will look at what mapping is exactly, and how it works. Then we will look at how to use multiple LIDARS while mapping, and build a launch file that allows us to map our world, and save the map for later use. Finally, we will actually launch the simulation from the launch file we just built, and map the house.
It is not recommended to follow this tutorial before finishing the ROS basics tutorial.

6.2 What is mapping:

Mapping is the process of taking raw sensor data from the LIDARs, and using it to determine where obstacles are located around the robot. All these obstacles can be collated into one map, which can then be saved and later used for navigation. This allows the robot to have knowledge of parts of the world that are beyond it's sensor range, and plan accordingly.

6.3 How does mapping work:

The gmapping algorithm we will be using works by reading the data from the laser scan topic, as well as the position and orientation of the robot. It combines this data to work out where the sensor readings are coming from in the world, and adds all the data into one map, which is saved as a map.yaml file.

6.4 IRA_laser_tools:

On this robot, we want to use two LIDARs, to be able to see around the manipulator that is in the middle of the body. However, the mapping algorithm we are using only takes input from one laser scan topic. To solve this problem, a ROS package was made called ira laser tools.
This package allows us to take data from multiple laser scan topics, and merge it into one output topic called scan that the mapping algorithm can use.
First we need to download the package, and put it in its own directory in the catkin_ws/src directory. Then, rebuild the catkin workspace with catkin_make.

Navigate to the launch folder and open “laserscan_multi_merger.launch”.
We want to edit the input topics to /scan1 and /scan2 (these are the topic names we defined earlier), and edit the output topic to be /scan, since this is the default topic gmapping listens to.

```
<param name="laserscan_topics" value ="/scan1 /scan2"
<param name="scan_destination_topic" value="/scan"/>
```

6.5 Creating the launch file:

The last step for us to start mapping is to create a launch file that will launch all the required ROS nodes, with the proper parameters.

Navigate to the launch directory in the example_urdf package, and create a file called mapping.launch:

```
$ cd ~/catkin_ws/src/example_urdf/launch
$ code mapping.launch
```

In this file, add the following:

```
<launch>
 <!-- Arguments -->
 <arg name="set_base_frame" default="base_footprint"/>
 <arg name="set_odom_frame" default="odom"/>
```

```
 <arg name="set_map_frame"  default="map"/>

 <!-- Gmapping -->
    <node    pkg="gmapping"    type="slam_gmapping"    name="robot_slam_gmapping"
output="screen">
   <param name="base_frame" value="$(arg set_base_frame)"/>
   <param name="odom_frame" value="$(arg set_odom_frame)"/>
   <param name="map_frame"  value="$(arg set_map_frame)"/>
   <param name="map_update_interval" value="0.01"/>
   <param name="maxUrange" value="5.0"/>
   <param name="sigma" value="0.05"/>
   <param name="kernelSize" value="1"/>
   <param name="lstep" value="0.05"/>
   <param name="astep" value="0.05"/>
   <param name="iterations" value="5"/>
   <param name="lsigma" value="0.075"/>
   <param name="ogain" value="3.0"/>
   <param name="lskip" value="0"/>
   <param name="minimumScore" value="50"/>
   <param name="srr" value="0.1"/>
   <param name="srt" value="0.2"/>
   <param name="str" value="0.1"/>
   <param name="stt" value="0.2"/>
   <param name="linearUpdate" value="1.0"/>
   <param name="angularUpdate" value="0.2"/>
   <param name="temporalUpdate" value="0.5"/>
   <param name="resampleThreshold" value="0.5"/>
   <param name="particles" value="100"/>
   <param name="xmin" value="-10.0"/>
   <param name="ymin" value="-10.0"/>
   <param name="xmax" value="10.0"/>
   <param name="ymax" value="10.0"/>
   <param name="delta" value="0.05"/>
   <param name="llsamplerange" value="0.01"/>
   <param name="llsamplestep" value="0.01"/>
   <param name="lasamplerange" value="0.005"/>
   <param name="lasamplestep" value="0.005"/>
```

```
 </node>
</launch>
```

6.6 mapping:

Now that we have set up our environment and launch files, we will start the mapping process. For this, we will need to launch all the launch files we made.
Run the following commands in separate terminals in order:

```
$ roslaunch example_urdf gazebo.launch
```

```
$ roslaunch ira_laser_tools laserscan_multi_merger.launch
```

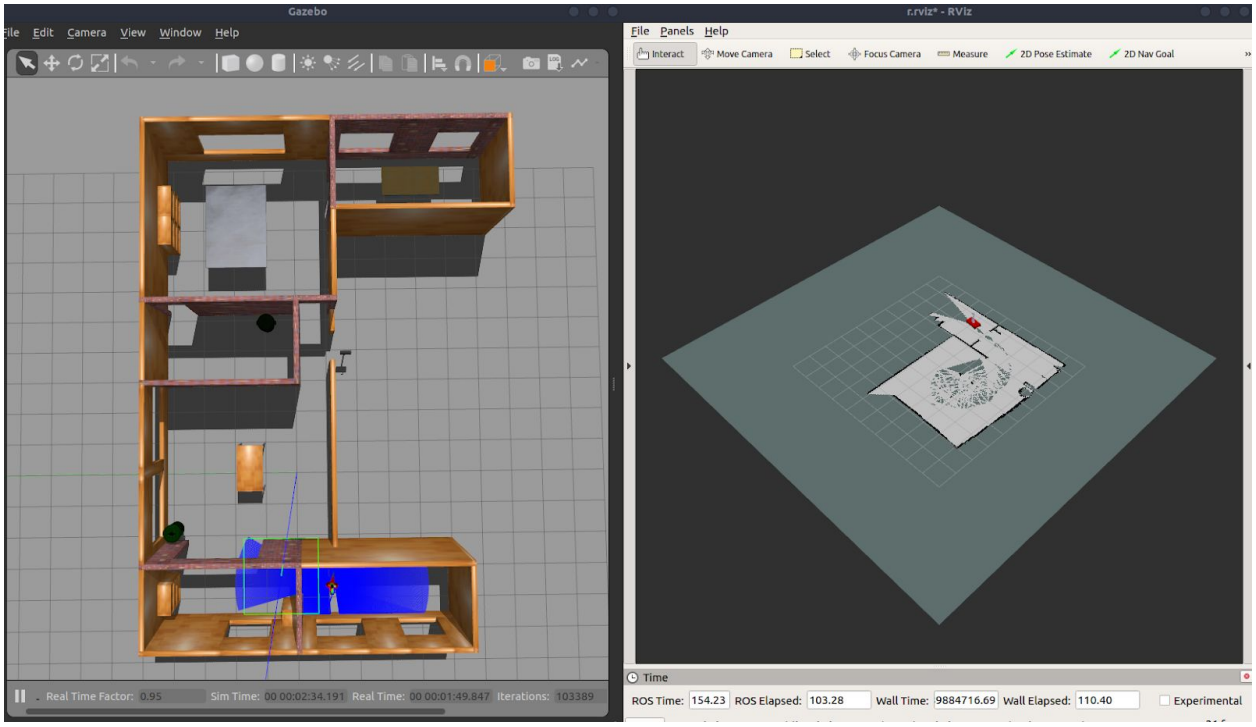
```
$ roslaunch example_urdf mapping.launch
```

In the rviz display panel click *add -> by topic -> map*
Then, in the global options change the fixed frame to map

Then, in a new terminal:

```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

You can now start navigating the robot around the environment. Notice that when you do so, the map is being updated, as visualised in rviz.



In order to get a high quality map, it can be necessary to do multiple passes. When done mapping we will create a new folder in the package and call it maps, and save the map by running the following commands in a new terminal:

```
$ cd catkin_ws/src/example_urdf/
$ mkdir maps
$ cd maps/
$ rosrun map_server map_saver -f map
```

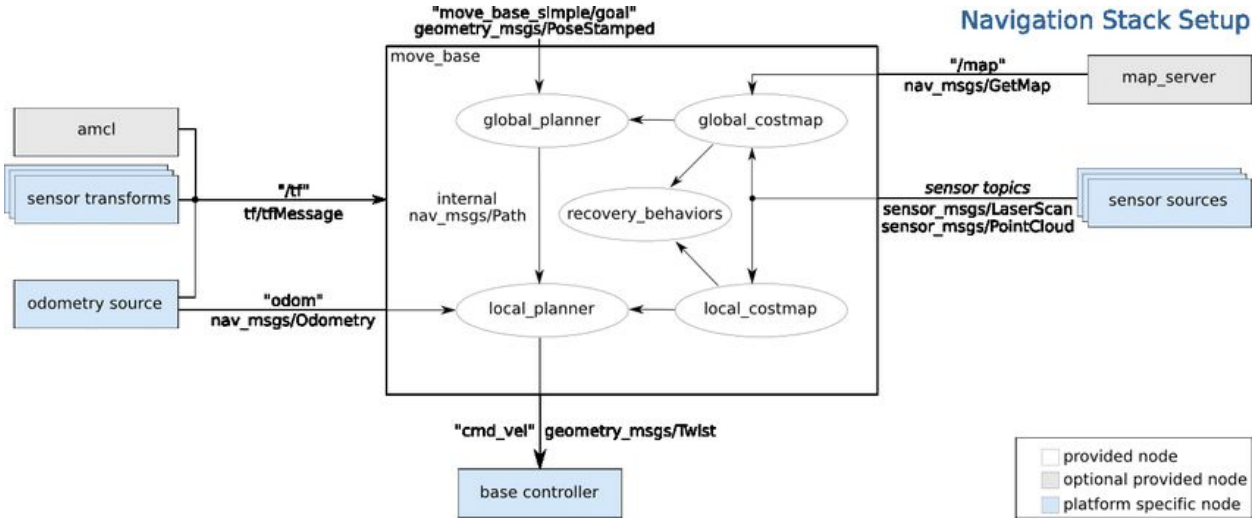
ROS will create 2 files, a .yaml and a .pgm file, which we will use later in the navigation tutorial. That will conclude this tutorial all files can be found in github.

7 . Navigation:

7.1 Intro

For the navigation of our robot, we will be making use of the ROS navigation stack.
The Navigation Stack is fairly simple on a conceptual level. It takes in information from odometry and sensor streams and outputs velocity commands to send to the mobile base. The navigation stack consists of two planners, a global and a local one. Both of these have their own

costmap. A costmap is a method of planning paths around obstacles. Areas close to obstacles, and paths through these areas, are assigned a higher 'cost' than those free from obstacles. In this tutorial, we will look at what these planners do, and tuning how exactly this costmap is made and evaluated, along with other parameters of the navigation stack.
Below is a graph that describes the architecture of the ROS navigation stack.



Looking at the graph we find that we already have the following;

Odom topic where gazebo is publishing odometry messages.

Scan topic that we created in the previous tutorial using ira laser tools.

A map that was created using ros map server.

To finish our navigation setup we still need to do the following:

Setting up a Local planner.

Setting up a Global planner.

Creating configuration files of the planners.

Creating launch files.

7.2 Global planner:

The global planner is responsible for looking at the map, and finding a path to reach the set navigation goal. In finding this path, the global planner only uses the map that was generated before, and not any additional sensor data. Adjusting to obstacles found by sensors that were not previously known is where the local planner comes in. In these tutorials, we will be using NavfnROS as a global planner, which is installed in the ROS workspace by default.

7.3 Local planner:

The local planner takes the path made by the global planner, and is responsible for actually executing it. This means that the local planner tries to find ways to move the robot that are close to the global path, while avoiding obstacles. The local planner we will be using in this tutorial is called Teb_local_planner. For more information about this package, please refer to the ROS wiki

Launch a terminal and install teb planner using the command:

$ sudo apt install ros-melodic-teb-local-planner

7.4 configuration files:

When using the navigation stack, move base has to be started with some parameters that will describe the behaviour of both global and local planners. These parameters are usually stored in .yaml configuration files.

These files have to be created in the navigation package, and called in the navigation launch file. you can download the files in github and copy them to the cfg folder in the example urdf package.

You can find a full description of the used parameters here.

7.5 launch files:

In this section we will make some minor adjustments to our gazebo.launch file, and create a teb.launch file.

Navigate to the launch folder in the example urdf package, and open gazebo.launch in an editor to apply the needed changes:

```
<?xml version="1.0" ?>
<launch>
 <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find example_urdf)/worlds/house.world"
/>
   <arg name="paused" value="false" />
   <arg name="use_sim_time" value="true" />
   <arg name="gui" value="true" />
   <arg name="headless" value="false" />
   <arg name="debug" value="false" />
```

```
 </include>
     <node     name="joint_state_publisher"     pkg="joint_state_publisher"
type="joint_state_publisher">
   <param name="use_gui" value="false" />
 </node>
  <param  name="robot_description"  command="$(find  xacro)/xacro  '$(find
example_urdf)/urdf/example_urdf.urdf.xacro'" />
     <node     name="robot_state_publisher"     pkg="robot_state_publisher"
type="robot_state_publisher" />
    <node    name="urdf_spawner"    pkg="gazebo_ros"    type="spawn_model"
respawn="false"  output="screen"  args="-urdf  -x  0  -y  0  -z  0    -model
example_urdf -param robot_description" />

                          <include                              file="$(find
ira_laser_tools)/launch/laserscan_multi_merger.launch" />
</launch>
```

Notice that we have removed the rviz node and added an ira_laser_tools node.

Now, still in the launch folder, create a teb.launch file, and add the following:

```
<launch>
   <param name="/use_sim_time" value="true" />
   <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
                                              <rosparam       file="$(find
example_urdf)/cfg/costmap_common_params.yaml"               command="load"
ns="global_costmap" />
                                              <rosparam       file="$(find
example_urdf)/cfg/costmap_common_params.yaml"               command="load"
ns="local_costmap" />
       <rosparam file="$(find example_urdf)/cfg/local_costmap_params.yaml"
command="load" />
                                              <rosparam       file="$(find
example_urdf)/cfg/global_costmap_params.yaml" command="load" />
                                              <rosparam       file="$(find
example_urdf)/cfg/teb_local_planner_params.yaml" command="load" />
       <param name="base_global_planner" value="navfn/NavfnROS" />
```

```
      <param name="planner_frequency" value="1.0" />
      <param name="planner_patience" value="5.0" />
                                    <param     name="base_local_planner"
value="teb_local_planner/TebLocalPlannerROS" />
      <param name="controller_frequency" value="5.0" />
      <param name="controller_patience" value="15.0" />
   </node>

   <!--  ****** Maps *****  -->
   <node name="map_server" pkg="map_server" type="map_server" args="$(find
example_urdf)/maps/map.yaml" output="screen">
      <param name="frame_id" value="map" />
   </node>

   <node pkg="amcl" type="amcl" name="amcl" output="screen">
            <rosparam  file="$(find  example_urdf)/cfg/amcl_params.yaml"
command="load" />
      <!-- We have a holonomic robot! Overwrite yaml config here! -->
      <param name="odom_model_type" value="omni" />
      <param name="initial_pose_x" value="0" />
      <param name="initial_pose_y" value="0" />
      <param name="initial_pose_a" value="0" />
   </node>

       <node   name="rviz"   pkg="rviz"   type="rviz"   args="-d   '$(find
example_urdf)/rviz/rviz_navigation.rviz'" />

</launch>
```

When launched, this file is going to start 4 nodes:

A move_base node with parameters stored in yaml configuration files.

A map_server nodes with the created map in the previous tutorial.

An amcl node to help localize the robot.

An Rviz node to visualise navigation paths and cost maps.

Now let's launch our navigation package:

```
$ roslaunch example_urdf gazebo.launch
$ roslaunch example_urdf teb.launch
```

Now we can send navigation goals through rviz using the '2D Nav Goal' button in the toolbar. Go ahead and test yours by doing so!
Depending on the goal you have selected the robot might collide with an obstacle or act in a different way than expected. In the next section we will tune the planner parameters to achieve the behaviour we want,using the RQt reconfigure plugin.

7.6 Tuning the navigation stack parameters:

7.6.1 Footprint:

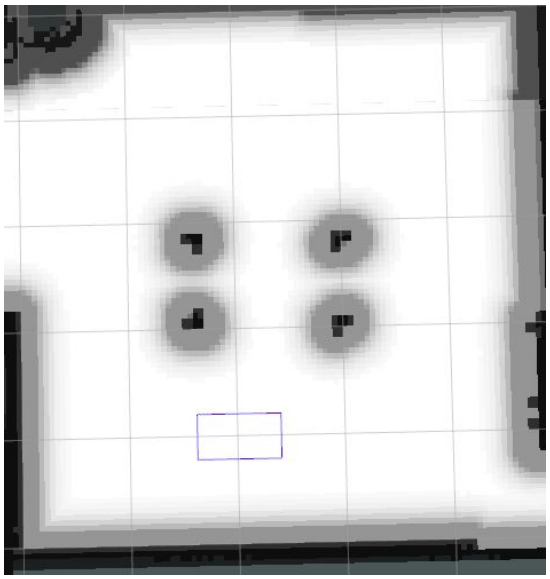
When you start the package for the first time, you'll notice that the robot is represented as a circle in rviz. This is because the robot's size is defined by the robot radius parameter in costmap_common_params.yaml. Let's override that parameter by adding the following line in the file:

footprint: [[-0.36,-0.2],[0.4,-0.2],[0.4,0.2],[-0.36,0.2]]

This will create a rectangle that will represent the robot footprint and will be used in the path planning.

7.6.2 Local cost map:

Notice the inflation, the area around an obstacle in the cost map, is quite large. For example the inflation around the table legs is .



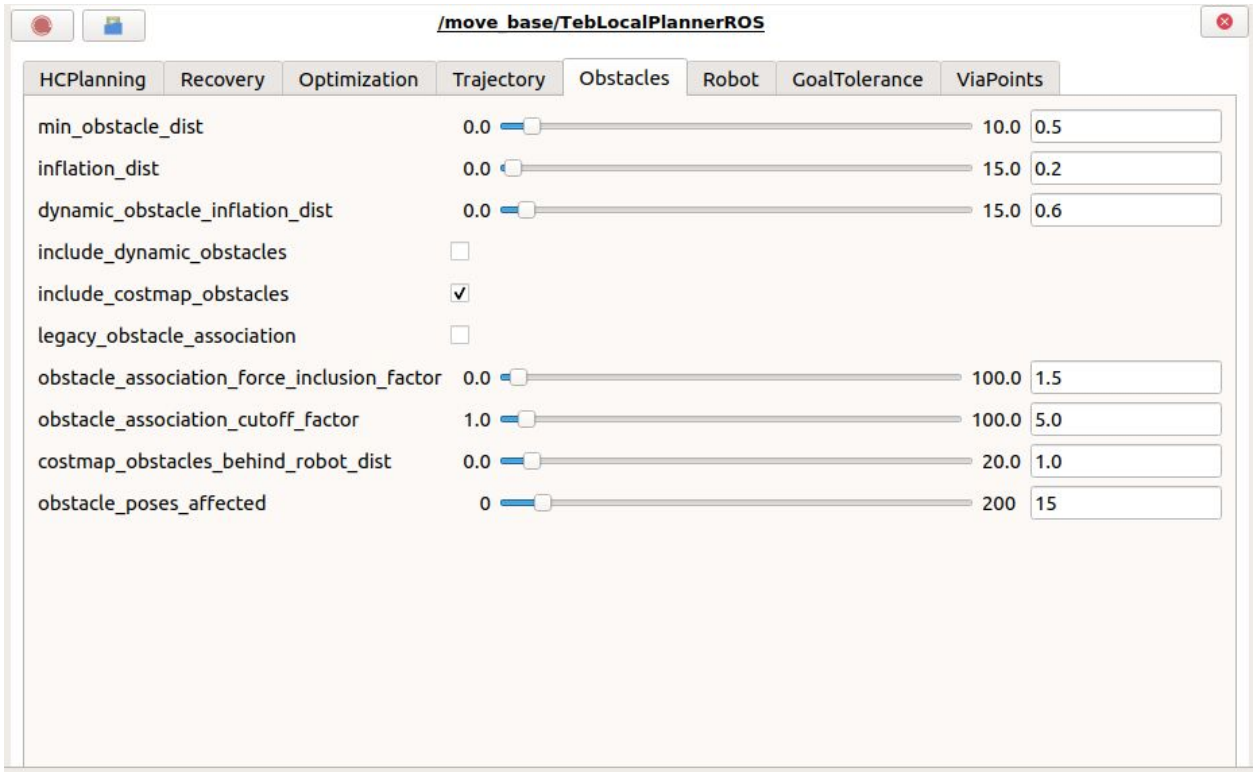
This can be changed in the same file under the inflation layer section.

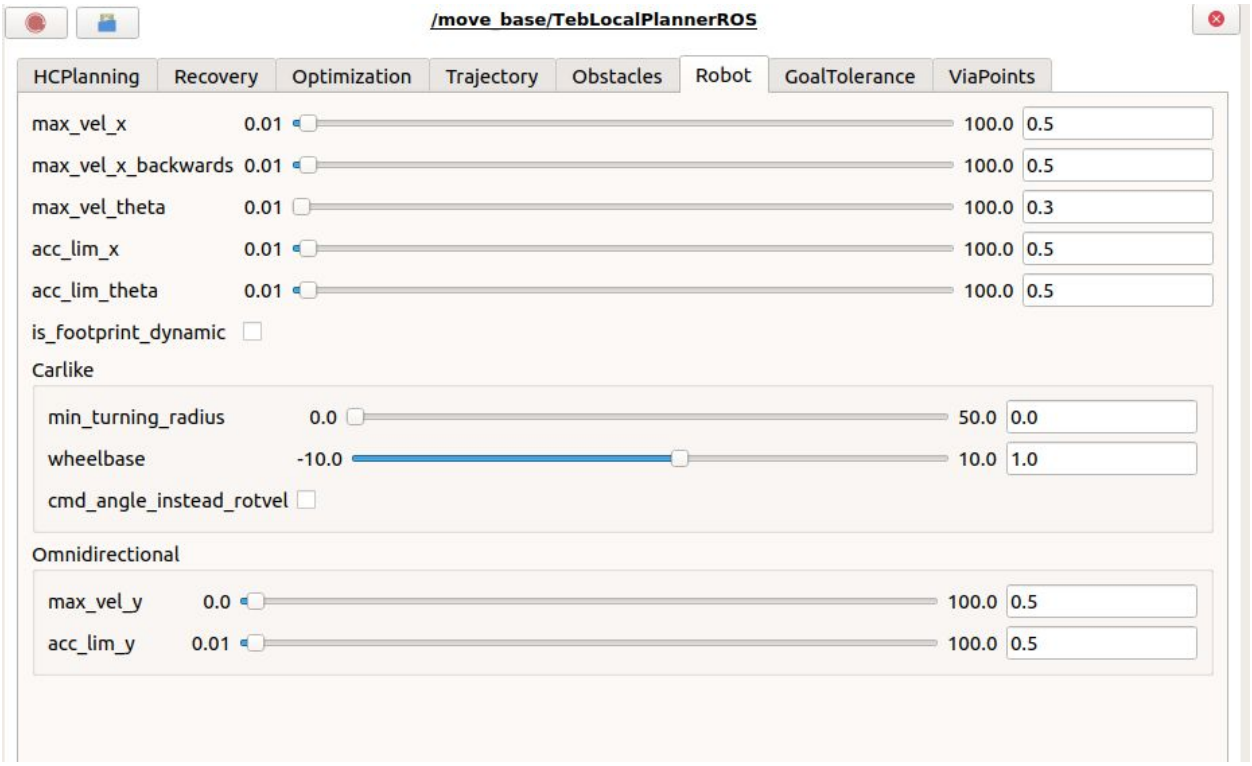
With every change that we are making in the configuration files we will have to restart the package again, that is when rqt_reconfigure becomes crucial in the tuning process.
start a new terminal and run the following command:

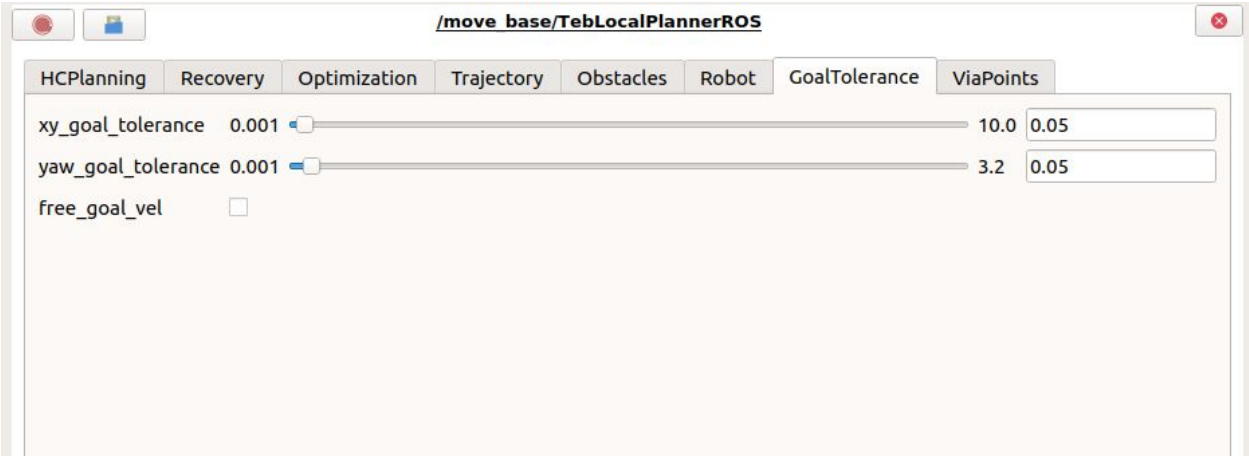
```
$ rosrun rqt_reconfigure rqt_reconfigure
```

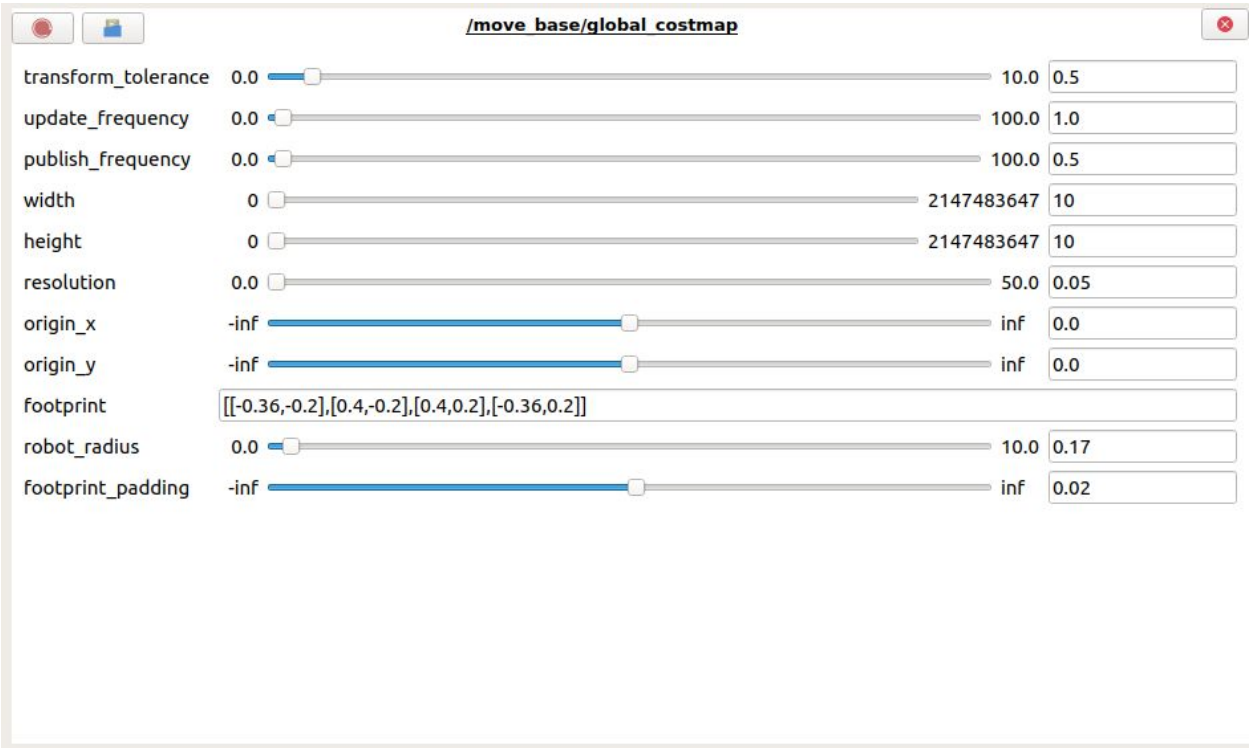
In the left panel select move_base -> global_costmap -> inflation_layer and change inflation radius value to 0.2 which will change the distance of inflation to 20 cm. Notice the costmap visualization in rviz becomes smaller.

Before copying our parameters please go ahead and try tuning the parameters yourself, read the effect of each parameter and experiment with changing it.
Below are the tuned parameters using rqt_reconfigure notice that you will have to write the changes to the yaml files for permanent change.
For more information about tuning teb planner please refer to [ros-wiki]




/move_base/TebLocalPlannerROS
HCPlanning
Recovery
Optimization
Trajectory
Obstacles
Robot
GoalTolerance
ViaPoints
max_vel_x 0.01 100.0 0.5
max_vel_x_backwards 0.01 100.0 0.5
max_vel_theta 0.01 100.0 0.3
acc_lim_x 0.01 100.0 0.5
acc_lim_theta 0.01 100.0 0.5
is_footprint_dynamic
Carlike
min_turning_radius 0.0 50.0 0.0
wheelbase -10.0 10.0 1.0
cmd_angle_instead_rotvel
Omnidirectional
max_vel_y 0.0 100.0 0.5
acc_lim_y 0.01 100.0 0.5





8 . ROS control

8.1 intro:

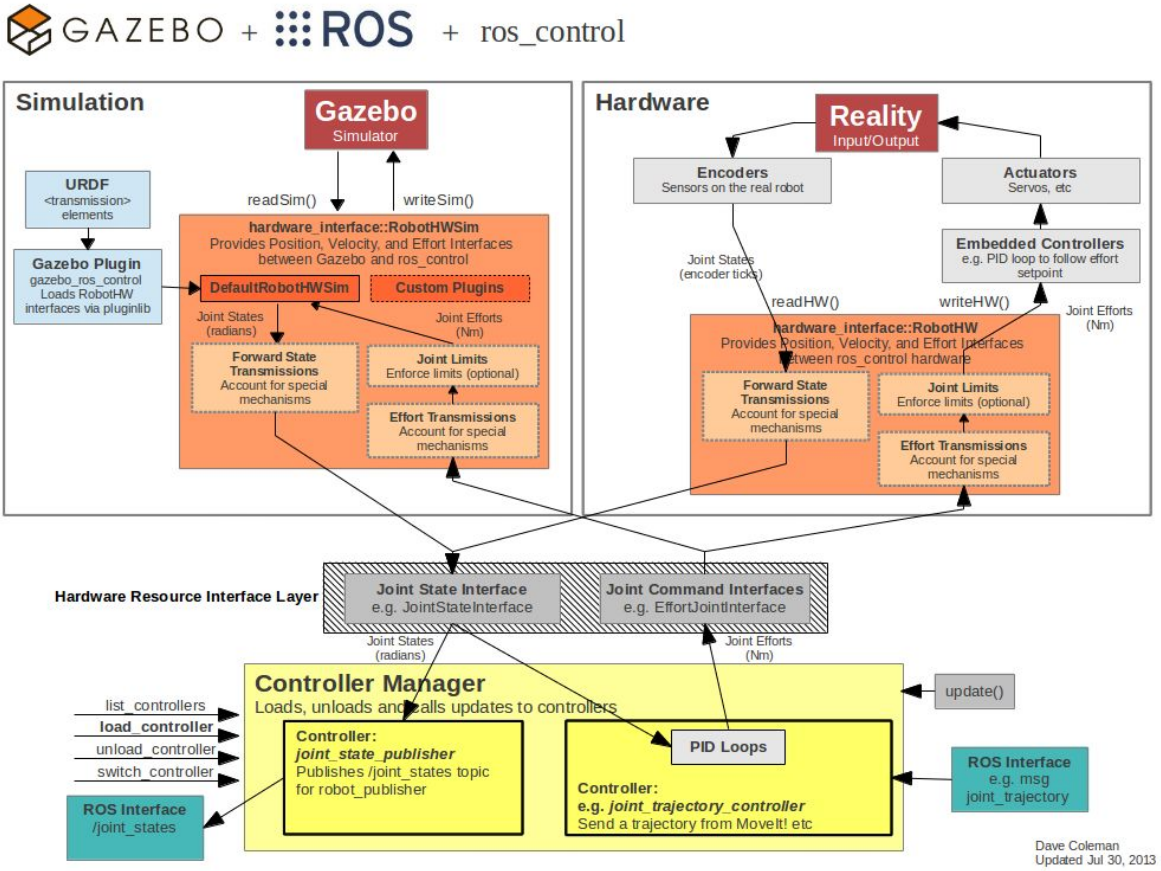
ROS control is a set of packages that include controller interfaces, controller managers, transmissions and hardware_interfaces. The mentioned packages are a rewrite of the pr2_mechanism packages to make controllers generic to all robots beyond just the PR2.

In other words ROS Control is the API that allows simple access to different actuators, and keeps the controller code separated from the actuator code.

In our case we are working with a simulated manipulator which will be controlled later using a planner. In order to make it possible for the planner “moveit” to control the manipulator we have to provide a ROS interface.
We will be using the ros_control packages which is a standard in ROS for controller interfaces. Please read the documentation for a better overview on [ROS control](#).

8.2 ROS Control & Gazebo:

The diagram below shows data flow of ros control and gazebo



More information can be found on [Gazebo sim](#).

8.3 transmissions:

To use ros_control with the robot we need to add <transmissions> to the urdf which will link virtual actuators to joints:

Download the urdf file of this tutorial from the github.

Navigate to example_urdf package -> urdf folder and edit example.gazebo.xacro by adding a transmissions for each joint at the end of the file:

```
<transmission name="tran0">
 <type>transmission_interface/SimpleTransmission</type>
 <joint name="jarmLink0">

<hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInte
rface>
 </joint>
 <actuator name="motor0">

<hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInte
rface>
   <mechanicalReduction>1</mechanicalReduction>
 </actuator>
</transmission>
```

8.4 Gazebo Ros_control plugin:

Gazebo plugin is used to parse the transmission tags and load the hardware interfaces and the controller manager.
Add the plugin to the example.gazebo.xacro:

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

And that will conclude the urdf setup.

8.5 Ros_control package:

Last thing we have to do to finish setting up ros control with gazebo is to create a control package. The control package will include configuration.yaml and a launch file.
Navigate to catkin_ws/src and create a package by running the following command:

$ catkin_create_pkg example_control controller_manager joint_state_controller robot_state_publisher

Navigate to the folder and create config and a launch directory:
$ cd example_control
$ mkdir launch config

Navigate to config folder and create a config.yaml file:

```
example_urdf:
 joint_state_controller:
   type: joint_state_controller/JointStateController
   publish_rate: 50
arm_controller:
 type: velocity_controllers/JointTrajectoryController
   joints:   [jarmLink0,   jarmLink1,   jarmLink2,   jarmLink3,   jarmLink4,
jarmLink5]
 gains:
     jarmLink0: {p: 10,  d: 0.1, i: 0.1, i_clamp: 1}
     jarmLink1: {p: 10,  d: 0.1, i: 0.1, i_clamp: 1}
     jarmLink2: {p: 10,  d: 0.1, i: 0.1, i_clamp: 1}
     jarmLink3: {p: 10,  d: 0.1, i: 0.1, i_clamp: 1}
     jarmLink4: {p: 10,  d: 0.1, i: 0.1, i_clamp: 1}
     jarmLink5: {p: 10,  d: 0.1, i: 0.1, i_clamp: 1}
 allow_partial_joints_goal: true
```

Navigate to launch folder and create a control.launch file:

```
<launch>
     <rosparam      file="$(find      example_control)/config/config.yaml"
command="load" ns="example_urdf" />
  <node  name="controller_spawner"  pkg="controller_manager"  type="spawner"
respawn="false"  output="screen"  ns="/example_urdf"  args="arm_controller"
/>
   <node    name="robot_arm_state_publisher"    pkg="robot_state_publisher"
type="robot_state_publisher" respawn="false" output="screen">
   <remap from="/joint_states" to="/example_urdf/joint_states" />
 </node>
```

```
</launch>
```

Now launch the description package:
$ roslaunch example_urdf gazebo.launch
Then launch the control package:
$roslaunch example_control control.launch

In a new terminal run rostopic list. Notice that we have a list of new available topics starting with the namespace /example_urdf/arm_controller/.
Those topics will form the interface between Gazebo and moveit which we will go in depth in the next tutorial.
All files can be found on github
that concludes this tutorial

9 . Manipulator control using MoveIt!:

9.1 intro:

In this tutorial we will learn how to set up graphical motion planning using moveit. To use the GUI that moveit provides we will need to create a moveit configuration package, then we will interface the created package with the control package that we have created in the previous tutorial in order to control our robot in gazebo using moveit.
Moveit installation guide can be found here.
We will be using trac-ik inverse kinematics solver run the following command to install it:

$ sudo apt install ros-melodic-trac-ik

9.2 setup assistant:

- Start a new terminal and run the following command:
  $ roslaunch moveit_setup_assistant setup_assistant.launch
- Select create new moveit configuration package.
- Brows the urdf file provided in the example_urdf description package, if you have followed the previous tutorials the file can be found in the following path:
  /home/i/catkin_ws/src/example_urdf/urdf/example_urdf.urdf.xacro
- Click load files.
- In the left panel select self_collisions and click generate collision matrix.
- In the panel select planning groups and click add group

    - Set group name to “arm”
    - Select trac_ik kinematics plugin in Kinematic solver.
    - Select RRT as a default planner.
    - Click add joints and select jarmLink0 to jarmLink5
- In the panel select Robot Poses and click add pose to store 2 valid poses.
- Select ROS control and click auto add followJointTrajectory controllers for each planning group.
- Select author information and enter your name and email.
- Select configuration files, choose a path to generate the package “/home/i/catkin_ws/src/robot_moveit” and click the Generate package button then exit setup assistant.

9.3 Moveit package:

Navigate to the package folder and look at the generated files, try to get familiar with the launch and configuration files.
To test the package run the following command:

```
$ roslaunch robot_moveit demo.launch
```

Rviz will start with a preconfigured file in the left panel you will see a new section called motionPlanning.
You can test the package following some simple steps:

- Go to the planning tab and choose the <current> as start state.
- in Goal state you will find the poses that you have created in moveit setup assistant chose one of them.
- Click plan.
- Click execute.
- You can see if executing the path is successful in rviz 3D preview.

Run the robot in a simulation environment by launching the gazebo.launch file from example_urdf package and try planning and executing a new path. You will find that moveit is not able to control the simulated robot in gazebo which will lead us to our next section in this tutorial on how to interface moveit with gazebo.

9.4 Moveit & Gazebo:

In the previous tutorial we have created a control package which started several new topics. If you did not follow the previous tutorial please go ahead and do it before continuing this one.

In this section we will configure the moveit generated package to interface it with gazebo.
Navigate to robot_moveit/config and create a new file, call it controllers.yaml with the following content:

```
controller_list:
 - name: example_urdf/arm_controller
   action_ns: "follow_joint_trajectory"
   type: FollowJointTrajectory
   joints:
     - jarmLink0
     - jarmLink1
     - jarmLink2
     - jarmLink3
     - jarmLink4
     - jarmLink5
```

Notice that we are referring to the chosen namespace in the previous tutorial "example_urdf/arm_controller

Create a new file called joint_names.yaml with the following content:

```
controller_joint_names:     ['jarmLink0',     'jarmLink1',     'jarmLink2',
'jarmLink3', 'jarmLink4', 'jarmLink5' ]
```

Now that we have stored our joints and controllers lists in yaml files we need to link them to moveit launch files. Navigate to robot_moveit/launch and edit the launch file called example_urdf_moveit_controller_manager.launch.xml:

```
<launch>
   <rosparam file="$(find robot_moveit)/config/controllers.yaml"/>
   <param name="use_controller_manager" value="false"/>
   <param name="trajectory_execution/execution_duration_monitoring"
value="false"/>
   <param name="moveit_controller_manager"
value="moveit_simple_controller_manager/MoveItSimpleControllerManager"/>
</launch>
```

Create a new launch file in the same directory and call it example_moveit.launch with the following content:

```
<launch>
            <rosparam              comman="load"              file="$(find
robot_moveit)/config/joint_names.yaml" />
```

```
 <include file="$(find robot_moveit)/launch/planning_context.launch">
   <param name="load_robot_description" value="true" />
 </include>
 <node name="joint_state_publisher_moveit" pkg="joint_state_publisher"
type="joint_state_publisher">
   <param name="/use_gui" value="false" />
   <rosparam
param="/source_list">[/example_urdf/arm_controller]</rosparam>
 </node>
 <include file="$(find robot_moveit)/launch/move_group.launch">
   <arg name="publish_monitored_planning_scene" value="true" />
 </include>
 <include file="$(find robot_moveit)/launch/moveit_rviz.launch">
   <param name="config" value="true" />
 </include>
</launch>
```

Now we have linked the moveit package to gazebo we can test it as follows:

- $ roslaunch example_urdf gazebo.launch
- $ roslaunch example_control control.launch
- $ roslaunch robot_moveit example_moveit.launch
- In rviz add motionPlanning.
- Planning and executing a path in rviz now will control our simulated robot in gazebo.

All files can be found in [github]

10 . Cartesian path planning in moveit:

10.1 Intro:

In this tutorial we will create a C++ package which will allow us to plan a cartesian path for the manipulator using an array of waypoints that will form this path.
To store the points we will use a JSON file. Using a Json file will allow us to change the path without having to change the code or recompile it, which will form a kind of interface between our package and a third party application.

10.2 Creating the package:

Create a catkin package called mov with roscpp dependency then edit the package.xml file and add the following:

```
<buildtool_depend>catkin</buildtool_depend>
<buildtool_depend>roscpp</buildtool_depend>
<build_depend>pluginlib</build_depend>
<build_depend>eigen</build_depend>
<build_depend>moveit_core</build_depend>
<build_depend>moveit_ros_planning</build_depend>
<build_depend>moveit_ros_planning_interface</build_depend>
<build_depend>moveit_ros_perception</build_depend>
<build_depend>interactive_markers</build_depend>
<build_depend>geometric_shapes</build_depend>
<build_depend>moveit_visual_tools</build_depend>
<build_depend>pcl_ros</build_depend>
<build_depend>pcl_conversions</build_depend>
<build_depend>rosbag</build_depend>
<build_depend>tf2_ros</build_depend>
<build_depend>tf2_eigen</build_depend>
<build_depend>tf2_geometry_msgs</build_depend>


<exec_depend>pluginlib</exec_depend>
<exec_depend>moveit_core</exec_depend>
<exec_depend>moveit_commander</exec_depend>
<exec_depend>moveit_fake_controller_manager</exec_depend>
<exec_depend>moveit_ros_planning_interface</exec_depend>
<exec_depend>moveit_ros_perception</exec_depend>
<exec_depend>interactive_markers</exec_depend>
<exec_depend>moveit_visual_tools</exec_depend>
<exec_depend>joy</exec_depend>
```

```
<exec_depend>pcl_ros</exec_depend>
<exec_depend>pcl_conversions</exec_depend>
<exec_depend>rosbag</exec_depend>
<exec_depend>tf2_ros</exec_depend>
<exec_depend>tf2_eigen</exec_depend>
<exec_depend>tf2_geometry_msgs</exec_depend>
```

Then edit the cmakelists.txt file by adding the following:

```
find_package(catkin REQUIRED
  COMPONENTS
        interactive_markers
        moveit_core
        moveit_visual_tools
        moveit_ros_planning
        moveit_ros_planning_interface
        moveit_ros_perception
        pluginlib
        geometric_shapes
        pcl_ros
        pcl_conversions
        rosbag
        tf2_ros
        tf2_eigen
        tf2_geometry_msgs
)

find_package(Eigen3 REQUIRED)
find_package(Boost REQUIRED system filesystem date_time thread)

set(THIS_PACKAGE_INCLUDE_DIRS
  doc/interactivity/include
)

catkin_package(
  LIBRARIES
  INCLUDE_DIRS
  CATKIN_DEPENDS
        moveit_core
```

```
        moveit_visual_tools
        moveit_ros_planning_interface
        interactive_markers
        tf2_geometry_msgs
  DEPENDS
        EIGEN3
)
```

Create a c++ file in the package src directory and call it move.cpp

Last step creating a launch file with the following content:

```
<launch>
<node     name="move_group_interface_tutorial"     pkg="mov"     type="move"
respawn="false" output="screen">
 </node>
</launch>
```

10.3 C++ & Json:

There are many different available libraries for C++ to parse JSON files. In this tutorial we will be using jsoncpp.so which is available by default in linux so it's not required to install or compile the library.

In our example we'll be reading a premade JSON file. Inorder to setup the part of the code related to JSON we have to include the following libraries:

```
#include <iostream>
#include <jsoncpp/json/value.h>
#include <jsoncpp/json/json.h>
#include <fstream>
```

Then we will declare some variables to be used to read the poses from the file:

A temp variable to hold the JSON values and parse them, then an array to store the parsed values.

```
geometry_msgs::Pose tempPose;
std::vector<geometry_msgs::Pose> waypoints;
```

A Json reader, value and an std::ifstream variable to read the file.

```
Json::Reader reader;
 Json::Value root;
 std::ifstream file("/home/i/catkin_ws/src/mov/src/points.json");
```

Then we will copy the file content to JSON value:

```
 file >> root;
```

And finally we will create a loop to read all the objects in JSON file and parse them to C++ double values then store all those positions in an array which will be used later.

```
for (int i = 0; i < root.size(); i++)
 {
   //parse tempPose
   tempPose.position.x = root[i]["px"].asDouble();
   tempPose.position.y = root[i]["py"].asDouble();
   tempPose.position.z = root[i]["pz"].asDouble();
   tempPose.orientation.x = root[i]["ox"].asDouble();
   tempPose.orientation.y = root[i]["oy"].asDouble();
   tempPose.orientation.z = root[i]["oz"].asDouble();

   waypoints.push_back(tempPose);
 }
```

10.3 Cartesian path planning:

To use the C++ interface include the following libraries:

```
 #include <moveit/move_group_interface/move_group_interface.h>
#include <moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>
#include <moveit_visual_tools/moveit_visual_tools.h>
#include <rviz_visual_tools/rviz_visual_tools.h>
```

Then start the regular ros setup:

```
 ros::init(argc, argv, "moveit_robot");
```

```
 ros::NodeHandle node_handle;
 ros::AsyncSpinner spinner(1);
 spinner.start();
```

Initialize move group and visual tools:

```
rviz_visual_tools::RvizVisualToolsPtr visual_tools_;

 static const std::string PLANNING_GROUP = "arm";

 moveit::planning_interface::MoveGroupInterface
move_group(PLANNING_GROUP);

 moveit::planning_interface::PlanningSceneInterface
planning_scene_interface;

 const robot_state::JointModelGroup *joint_model_group =
     move_group.getCurrentState()->getJointModelGroup(PLANNING_GROUP);

 namespace rvt = rviz_visual_tools;
 moveit_visual_tools::MoveItVisualTools visual_tools("base_footprint");
 visual_tools.deleteAllMarkers();
 visual_tools_.reset(new
rviz_visual_tools::RvizVisualTools("base_footprint",
"/rviz_visual_markers"));
```

Then we parse the JSON file as we saw in the previous section
Then create a trajectory to store the path in.

```
 moveit_msgs::RobotTrajectory trajectory;
```

Create some variables to define the cartesian path constraints:

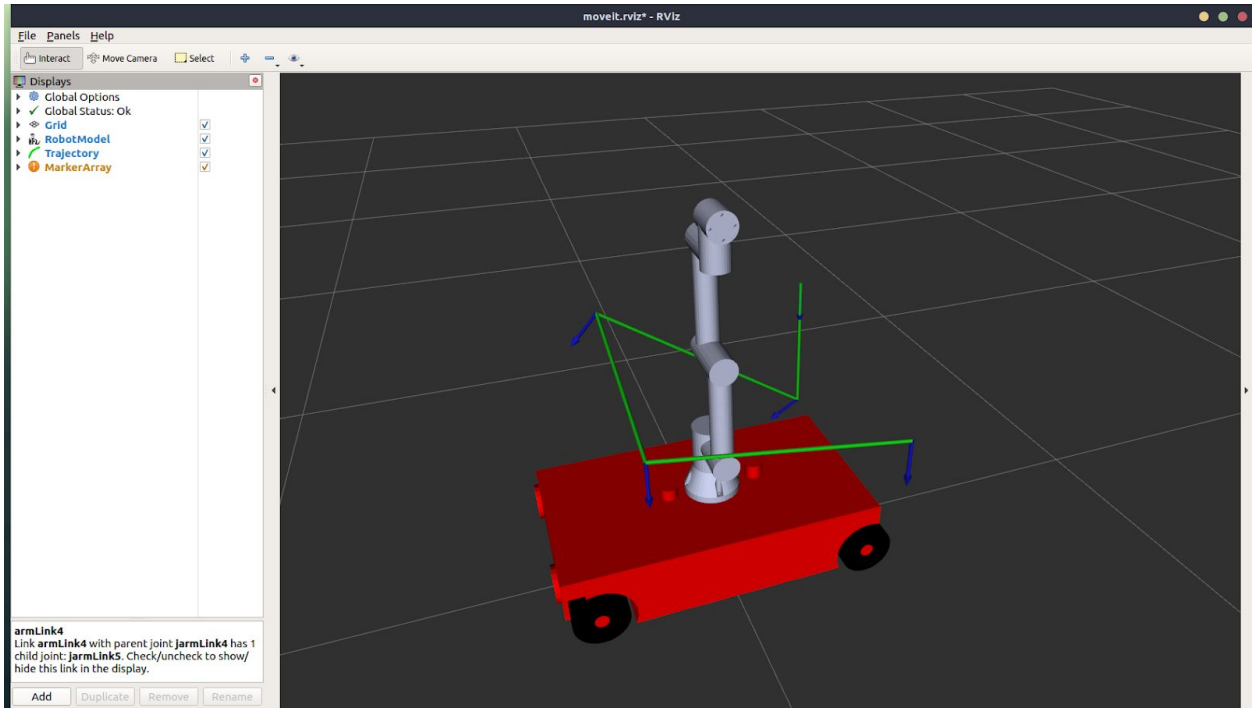
```
const double jump_threshold = 0.0;
 const double eef_step = 0.01;
```

Call the computeCartesianPath and pass the variables created earlier then use viausl_tools to visualize the results in rviz:

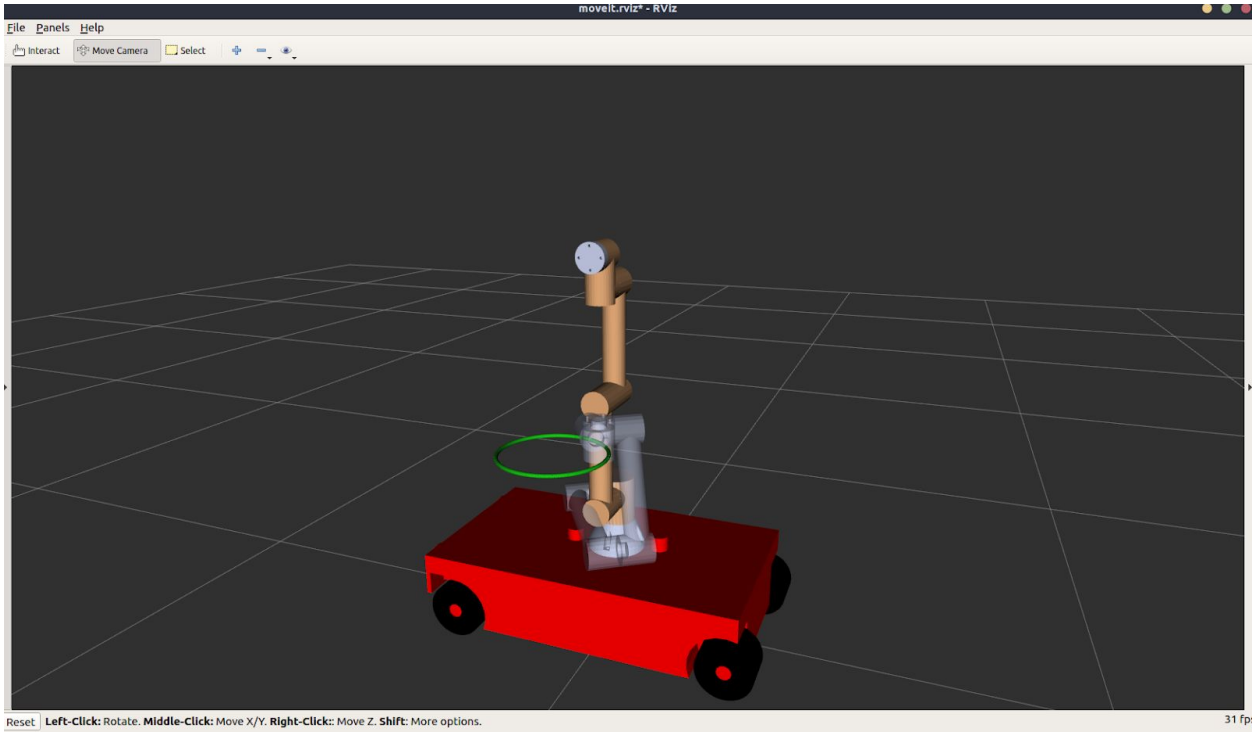
```
double fraction = move_group.computeCartesianPath(waypoints, eef_step,
jump_threshold, trajectory);
```

```
 ROS_INFO_NAMED("tutorial", "Visualizing plan 4 (Cartesian path) (%.2f%%
acheived)", fraction * 100.0);

 //visual_tools.deleteAllMarkers();
 visual_tools.publishPath(waypoints, rvt::RED, rvt::SMALL);
 for (std::size_t i = 0; i < waypoints.size(); ++i)
 {
   visual_tools.publishZArrow(waypoints[i], rviz_visual_tools::BLUE,
rviz_visual_tools::SMALL, 0.1);
 }
 visual_tools.trigger();
```

Launch the robot_moveit demo.launch file
Launch the mov move.launch package
Add the markers topic in rviz and you will be able to visualize the cartesian path and the end effector state at each waypoint from the json file.



You can keep experimenting with cartesian path planning in the git repo [github] You can find a c++ file called circularPath.cpp which defines an array of waypoints that forms a circle which will result in the following path:



That will conclude this tutorial all files can be found in [github]

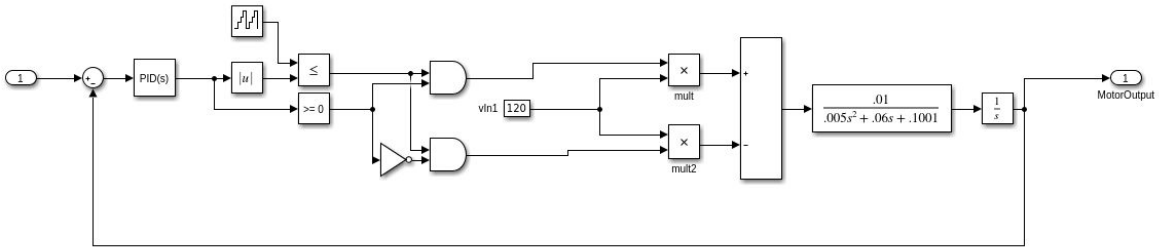
11 . Matlab and Ros integration:

11.1 intro:

This tutorial will show how to integrate a matlab generated code for a motor pid controller with ros , we will use a simple two link robot to visualize the results in rviz, the robot package can be downloaded at [github].

11.2 JSP node:

As mentioned before the code is generated from a simulink model which is included below



This model takes a setpoint as input and with each simulation step sends the position of the motor back.
The main generated function we will be using is called step.

After generating code from matlab take some time to read the code and get familiar with the objects and functions used in this code.
We have created a new c++ file and included an instance of the simulink model class called driver.
In the main function we are using the return value of the step function to edit a variable called msg of type sensor_msgs::JointState. Then we are publishing the values on a ros topic that is visualized in rviz.
The code:

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

 ros::NodeHandle nh;
 ros::Publisher pub =
nh.advertise<sensor_msgs::JointState>("joint_states", 1000);
 ros::Rate loop_rate(10000);

 sensor_msgs::JointState msg;
 msg.name.resize(1);
```

```
 msg.name[0] = "joint1";
 msg.position.resize(1);

 /*****************************************************************/
 finaldriverModelClass driver;
 driver.initialize();
 //driver.finaldriver_U.INPUT = 100; // a command

 for (size_t i = 0; i < 10; i++)
 {
   std::cin >> driver.finaldriver_U.INPUT;
   while (true)
   {
     driver.step();
     printf("%f \n", driver.finaldriver_Y.MotorOutput);
     msg.position[0] = driver.finaldriver_Y.MotorOutput;
     msg.header.stamp = ros::Time::now();

     pub.publish(msg);
     ros::spinOnce();
     loop_rate.sleep();
     if(sqrt((driver.finaldriver_Y.MotorOutput -
driver.finaldriver_U.INPUT)*(driver.finaldriver_Y.MotorOutput -
driver.finaldriver_U.INPUT)) < 0.001)
     {
       break;
     }
   }
   /*****************************************************************/
 }
 return 0;
}
```

Launch the my_robot_description package.
Launch the jsp node

Notice that the node takes terminal input from the user, with each value the two links robot will move to the setpoint that is used as user input in the jsp node.

And that will conclude integrating matlab generated code with ros.