TUM Institute for Cognitive Systems (ICS)
RoboCup@Home course

Tutorial 2: Simulations with Gazebo and ROS

M. Sc. Rogelio Guadarrama

30.10.2019

In this tutorial, a number of tools to set up simulation environments will be provided. The first sections are dedicated to describe the main features of the Gazebo simulator and ros_control framework. Some step by step examples are provided to understand these features. In the last section of this document (section 6), a set of four exercises will be requested. These four exercises will be evaluated for the grade. Please type the commands instead of copying them from the PDF to prevent bad string formating in the linux terminal.

1 The Gazebo simulator

Gazebo is a powerful simulation environment created to provide close-to-reality scenarios to test robotic platforms safely and under controlled conditions. The Gazebo Project is constantly developed by the Open Source Robotics Foundation (OSRF). The ROS community has adopted Gazebo as a preferred simulation environment and since 2013, it has been the official platform of the Virtual Robotics Challenge, a component in the DARPA Robotics Challenge.

Figure 1.1: More information about the Gazebo project in *http://gazebosim.org*

Gazebo 7.12 is provided within the installation of ROS-kinetic as well as a number of models and examples. However, there are a lot more models of furniture, buildings and robots provided by the ROS community and the Robot manufacturers. The platform assigned for this course is the TIAGo robot from PAL robotics, in the next sections, the set-up procedure will be described as well as some exercises to stand out some key features and tools of Gazebo simulator and TIAGo framework.

2 Simulation Setup.

In this practical course, we will use two different robots: TIAGo from PAL robotics and HSRB from Toyota. Keep in mind that they use different package versions, therefore you will need to have two independent workspaces one to work with TIAGo and one for the HSRB robot. The software to run the HSRB robot is already installed in the lab's workstations.

2.1 Preparing the TIAGo simulation environment.

The software to run TIAGo on simulation is provided for free by PAL Robotics. The model, the hardware interface and a number of examples coded in C++ and Python are also provided at: *http://wiki.ros.org/Robots/TIAGo/Tutorials*

2.1.1 Create a workspace to run the simulation.

The required packages to run the simulation are specified at

`http://wiki.ros.org/Robots/TIAGo/Tutorials/Install`

All these packages are already installed in the pc of the laboratory in order to save time. First, create a workspace named as *~/ros/worspace/tiago_ws_YOURNAME*. Then, open a terminal and run the commands below to create the folder where the packages will be downloaded.

```
$ cd ~/ros/worspace/tiago_ws_YOURNAME/src
$ mkdir tiago
```

2.1.2 Populate the workspace

Locate the file *tiago_public.rosinstall* file from the provided material. It can be found at:

https://raw.githubusercontent.com/pal-robotics/tiago_tutorials/master/tiago_public.rosinstall

move the file to the *~/ros/worspace/tiago_ws_YOURNAME/src/tiago* directory. Make sure that the file has executable permission. Then run the next command and wait until the cloning process finishes.

```
$ cd ~/ros/worspace/tiago_ws_YOURNAME/src/tiago
$ rosinstall . tiago_public.rosinstall
```

Once the folders are copied into *src* directory, compile the new packages, running the flag -DCATKIN_ENABLE_TESTING=0 the first time

```
$ cd ~/ros/worspace/tiago_ws_YOURNAME/
$ catkin_make –DCATKIN_ENABLE_TESTING=0
$ source devel/setup.bash
```

It may be needed to run it more than one time if errors are showed. After the compilation process reaches a *100%* and finishes, close the terminal.

2.1.3 Testing the installation

Once the workspace is compiled, a simulation environment from the ones provided by PAL can be launched to test the installation. Open two terminals and run the commands below to enter the workspace and source the built project on each terminal. These two commands must be executed every time you open a terminal to work with the tiago workspace.

```
$ cd ~/ros/worspace/tiago_ws_YOURNAME
$ source devel/setup.bash
```

On the first terminal, run the next command.

```
$ roslaunch tiago_gazebo tiago_gazebo.launch public_sim:=true robot:=steel world:=tutorial_office
```

A Gazebo GUI window should be displayed showing the TIAGo Robot in a office. The robot will perform an initialization routine to get the default home position.

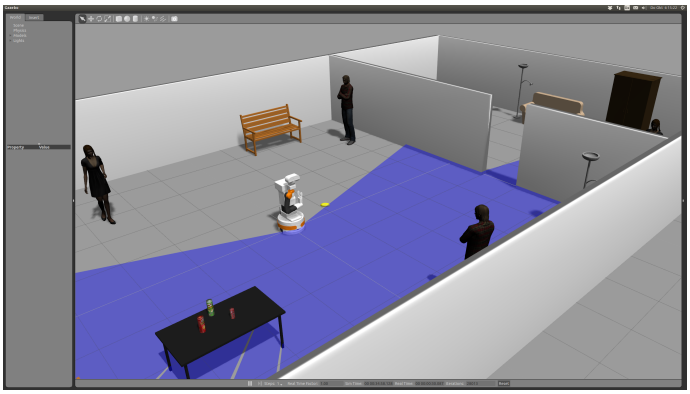

Figure 2.1: Default home position of TIAGo robot.

Once the robot reached the home position, run the next command on the other terminal to execute a demo for the mobile base.

```
$ rosrun key_teleop key_teleop.py
```

With the *key_teleop* demo window active, use the arrow keys to navigate with the robot trough the office. To finish the simulation, press *q* on the *key_teleop* terminal and *ctrl (strg)* + *c* on the Gazebo terminal.

2.2 Preparing HSRB simulation environment.

First, make sure that the HSRB simulation packages are installed. Then, create one workspace for the HSRB robot named as *~/ros/worspace/hsrb_ws_YOURNAME*. Open three terminals and source the workspace in a similar way as the tiago workspace.

```
$ cd ~/ros/worspace/hsrb_ws_YOURNAME
$ source devel/setup.bash
```

On the first terminal, run the next command and wait until the Gazebo and the Rviz windows are loaded (see Fig. 2.2).

```
$ roslaunch hsrb_gazebo_launch hsrb_megaweb2015_world.launch
```

By default, the HSRB robot simulation is paused after launching. To start the dynamics engine, click on the *play* button at the bottom of the Gazebo window. Once the simulation is running, the topics should be visualized in the Rviz as shown in Figure 2.3. In another terminal run the *key_teleop* node.

```
$ rosrun key_teleop key_teleop.py
```

Now, remap the publisher of the *key_teleop* node to the topic where the HSRB robot is looking for velocity commands for the mobile base:

```
$ rosrun topic_tools relay /key_vel /base_velocity
```

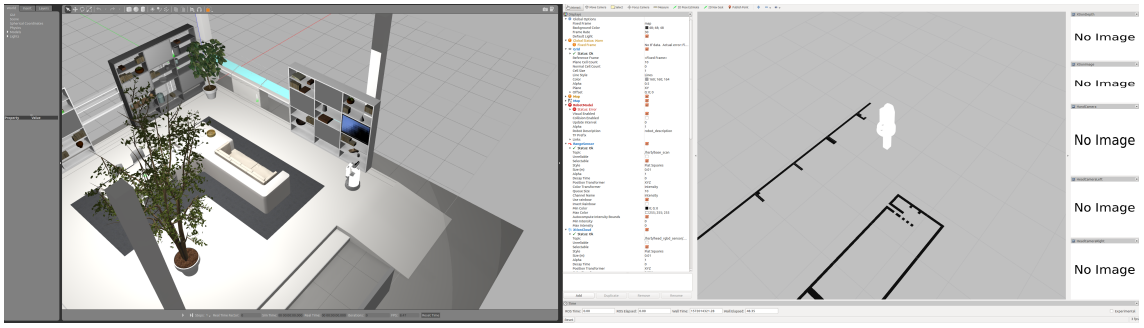

Figure 2.2: HSRB robot simulation after launched. Notice that the simulation starts paused.

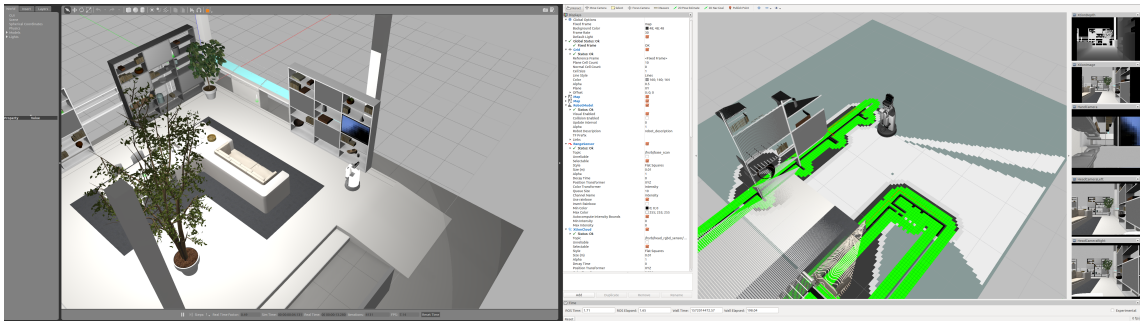

Figure 2.3: HSRB robot simulation running.

With the *key_teleop* window active, use the arrow keys to navigate with the robot trough the apartmentt. To finish the simulation, press *q* on the *key_teleop* terminal and *ctrl (strg) + c* on the other terminals.

3 Building up a scenario

Gazebo simulator can load models from the folders specified on the environment variables and also download open source models from *http://gazebosim.org/models/*. The default source of models is the directory:

```
~/.gazebo/models/
```

To work this section, chose one robot to work and source the corresponding workspace.

3.1 Download models from the server

1.- Create a folder named *tum_ics* in the *src* folder of your workspace and copy the *ics_gazebo* package inside. The *ics_gazebo* package is in the template folder of this tutorial.

2.- Find the *edit_world.launch* file in the *ics_gazebo* package and open it with a text editor. Follow the instructions in the file to add the model libraries depending on the robot environment you are using.

3.- Open a terminal, source your workspace and compile it again to include the new package in the environment. Then run the following command.

```
$ roslaunch ics_gazebo edit_world.launch
```

4.- On the Gazebo window, click on the *Insert* tab, scroll down and expand the model lists.

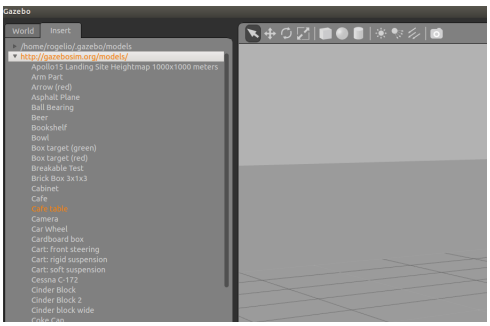

Figure 3.1: Model list on *http://gazebosim.org/models/*

5.- Click once on the word *table* from the list and then move the mouse over the simulation space. You will see that the model of a table is following the cursor. Click somewhere on the simulation space to place a table on the scenario. Repeat this step twice (see Fig. 3.2).

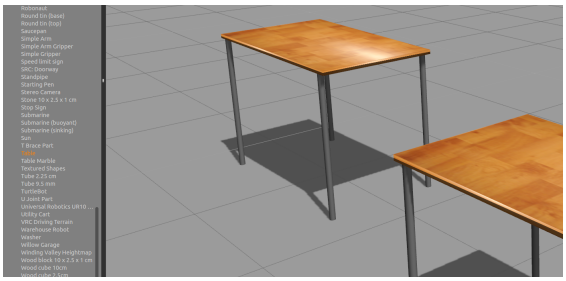

Figure 3.2: Place two tables.

6.- To remove a model from the simulation. Click on the *World* tab from the left panel. Then, expand the *Models* list (see Fig. 3.3). These are the models included in the simulation. Right click on one of the table names and click *Delete*.

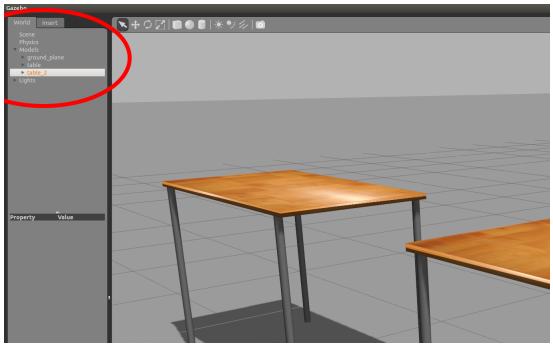

Figure 3.3: Removing one model.

3.2 Handling models in the simulation

Assume you want to change the position and orientation of an object in the simulation space. Follow the next steps to know the tools available on Gazebo to handle objects on the simulation space.

1.- Place a small object under the table, following the first steps of the subsection 3.1.

2.- With the simulation window active, press *T* on the keyboard. Then, click on the object under de table. You will see a coordinate frame attached to the object (see Fig. 3.4).

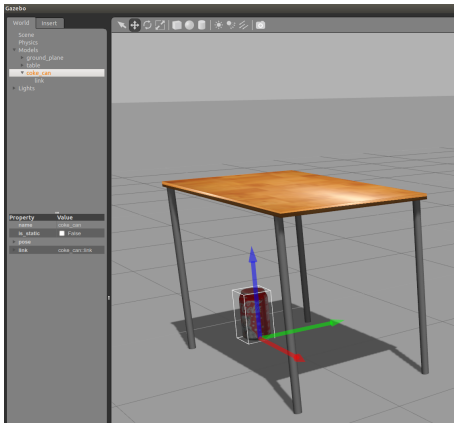

Figure 3.4: Moving an object on the simulation.

3.- Place the pointer of the mouse over the axis pointing up, then, click and grab the object to place it over the table.

4.- Repeat these steps with another small object.

5.- Press *R* on the keyboard and click on one of the objects to enable the orientation handle (see Fig. 3.5).

6.- Rotate the object click-grabbing on the orientation handle. If you have problems setting the position of an object, you can pause the simulation to disable the physics engine.

7.- To save the scenario click on the *File* tab from the window menu and then in *Save world as*. Save the world in the *ics_gazebo* package in the worlds folder.

`~/ros/worspace/ROBOT_ws_YOURNAME/src/ics_gazebo/worlds/`

Save the scenario in that directory with the name *tutorial.world*

NOTE: It is important that the world contains no robots at the moment it is being saved, otherwise the robot spawner may have issues when called. Therefore, any robot model should be removed from the models list from the world tab in the left panel before saving the world.

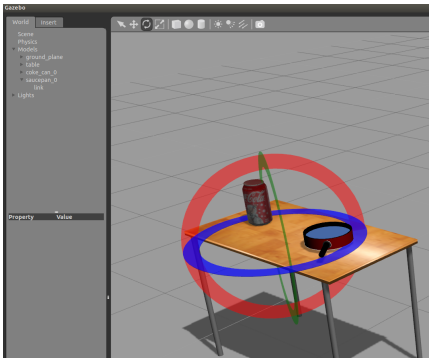

Figure 3.5: Rotating an object on the simulation.

3.3 Testing the scenario with the robot

To test the scenario. Launch the simulation with the robot and the *key_teleop* node similar to the previous section. Remember to remap the *key_vel* topic if you are using the HSRB robot.

1.- In one terminal, run the command below. The field *robot_pos* defines the initial position and orientation of the robot in meters for the *xyz* axes and radians for *Y* (Yaw). Replace *ROBOT* in the command with *tiago* or *hsrb* depending on the robot environment you are using.

```
$ roslaunch ics_gazebo ROBOT.launch world_suffix:=tutorial
  robot_pos:=-"x 0.0 -y 0.0 -z 0.0 -Y 0.0"
```

2.- In another terminal, launch the *key_teleop* node as shown in Section 2.

3.- After testing the scenario, finish the simulation as shown in Section 2.

4 Building up a new model

If the models available on the gazebo server and the models provided by PAL robotics or Toyota are not enough to build up a scenario, new models can be built. One model is composed by 3 files.

- A *model.config* file which contains information about the author of the model and a textual description of the model.
- A *.sdf* file containing all the physical properties of the model.
- Meshes of the body parts in *STL* or *collada* format.

4.1 Description of the files

SDF files are *xml* arrays used to describe all the components of a simulation. *xml* is a tree-oriented array file format commonly used in databases. For the gazebo simulator, the file must have at least the components showed on Fig 4.1.

More information on *SDF* in *http://sdformat.org/spec*

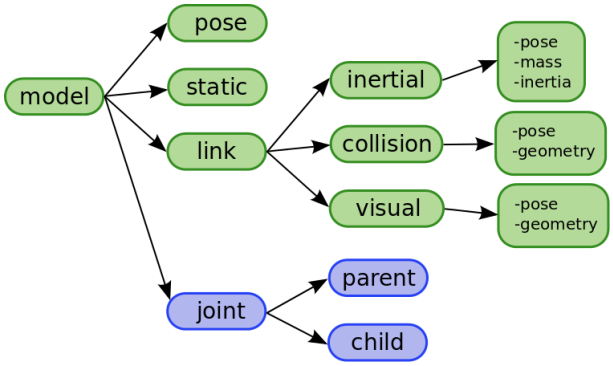

Figure 4.1: Principal components of a *sdf* file. Mandatory components are highlighted in green and optional components in blue.

1.- Locate the *models* folder in the *ics_gazebo* package. Inside that folder, the described files are contained for a couple of models. Open and read the file *model.config* in the *door* folder. Write your name as author of the model.

2.- Locate the file *door.sdf* and open it in a text editor.

3.- Open the Gazebo simulator to visualize the changes you do to the model. Run the following command in a terminal.

```
$ roslaunch ics_gazebo edit_world.launch
```

4.- Load the *Door* from the *ics_gazebo* package model list as shown in Section 3. The model should look completely wrong.

5.- The file *door.sdf* contains three links and three joints to describe the door. Follow the TODO tags to fix the file to contain the mandatory elements as well as some other optional components to build a movable frame-door-handle model.

6.- Every time you change something in the model file, save it and import it again in the Gazebo window to visualize the changes. You can keep the Gazebo window open all the time.

4.2 Testing the model

In order to see the created model on the simulation, use the steps described in section 3 to import the door model on a simulation with the robot and use the *key_teleop* node to pass through the door (see Fig. 4.2).

5 Gazebo + ROS + ros_control

At this point, scenarios can be prepared to test algorithms that will be provided on further tutorials. These algorithms will read data from the sensors of the robot and command actions to the motors. The bridge between the hardware and the software is created through ROS and the ros_control framework.

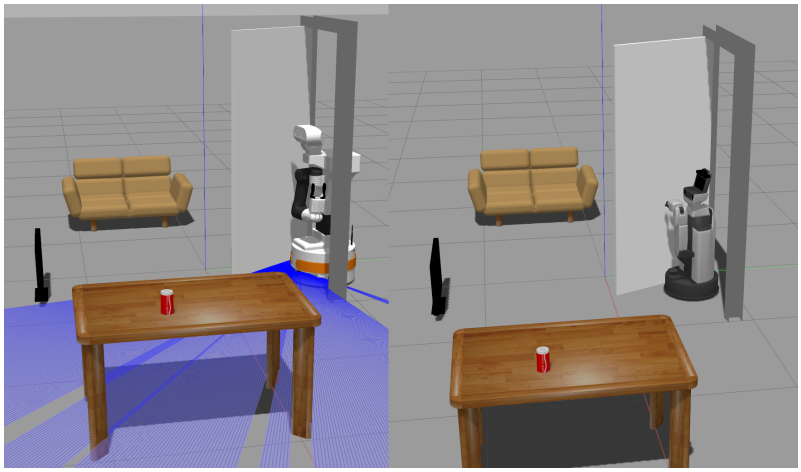

Figure 4.2: Passing through the door with the robots.

The ros_control framework is a developing tool intended to generalize controller applications and make them robot-agnostic. This general approach allows to share code between different platforms including simulated ones. Therefore, the code created using a simulated environment will be the same than the one used to deploy on a real robot.

This capability is enabled by the inclusion of a hardware-abstraction layer between the controller software and the hardware drivers. Such layer consist on a base class whose functions are overloaded with the needs of the hardware of the robot (or simulation). In Fig. 5.1 are shown the main components of the ros_control framework in a block diagram.

The main components of the ros_control framework are:

- **Hardware abstraction layer:** The bridge between hardware and software.
- **Controller Manager:** A ROS node provided to load controllers as plugin-libraries and connect them to the hardware abstraction layer. The controller manager is commanded through service calls. The controller manager also provides a set of default controllers to command the joints by publishing messages to a topic and to read the sensors subscribing to a topic.
- **Controller plugins:** Plugin libraries written to perform a specific task on a number of resources (sensors and actuators). These libraries must be created using the hardware interface header classes and the base_controller Class. Several controller plugins may be loaded at the same time but they must not try to access the same actuators. Therefore, two controller plugins can read the same sensor at the same time but not command the same actuator.

5.1 Reading data from the sensors

When the simulation environment is launched (as in section 2.1.3), a number of nodes are launched and also a number of topics where the states of the robot and data from the sensors becomes available for subscribers. The rviz tool is helpful to visualize all the published messages.

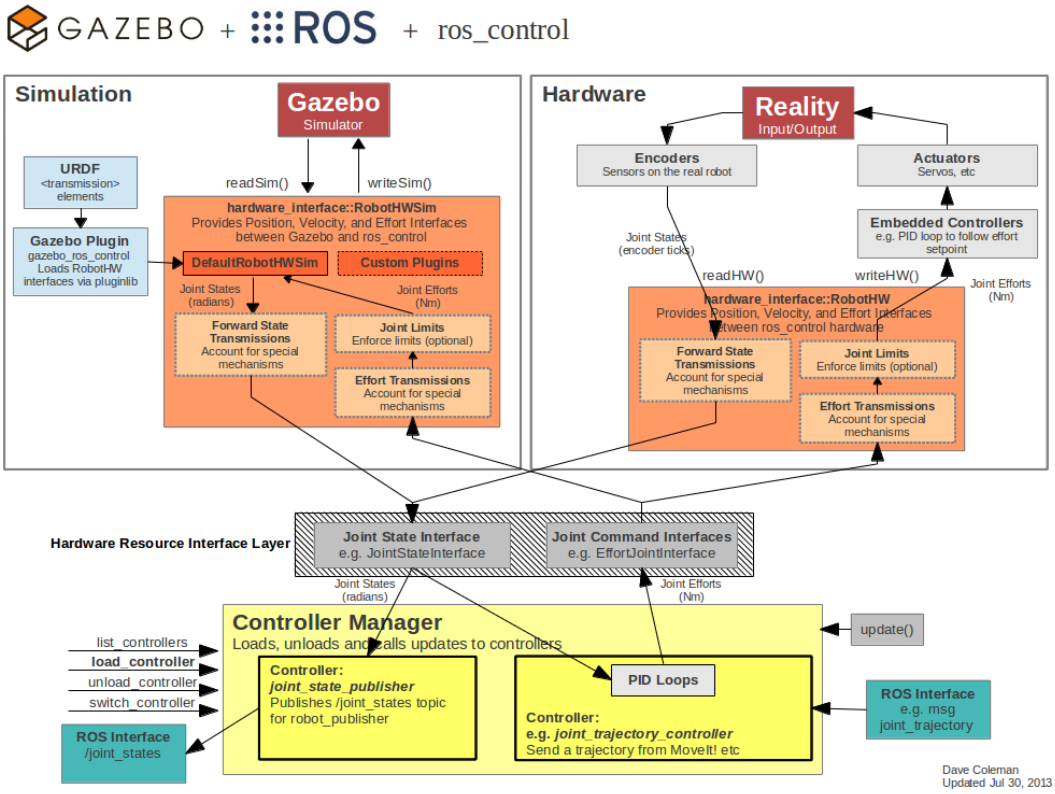

Figure 5.1: Block diagram of the ros_control framework. The hardware abstraction layer is highlighted in gray color. The implementation of the low level functions are highlighted in orange blocks, these blocks are commonly provided by robot manufacturers. The controller manager (highlighted in yellow) is a node in charge of loading and managing the controller plugins.

1.- Launch the simulation of a robot in one a world with objects.

```
$ roslaunch ics_gazebo ROBOT.launch world_suffix:=tutorial
```

2.- After the rviz window is loaded, in the panel at the left side, set the *Fixed Frame* property to the *base_footprint* frame.

3.- At the bottom of the same panel click on the *Add* button. Then, from the opened window select *RobotModel* to visualize the robot. You can also add a *Grid* element to visualize the floor (see Fig. 5.2).

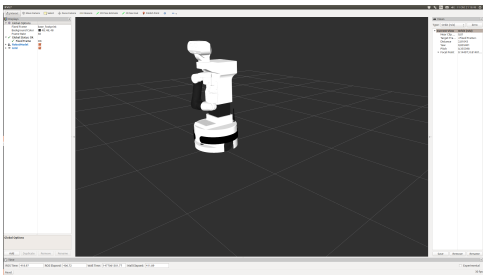

Figure 5.2: TIAGo robot displayed in rviz.

4.- The ultrasonic range finders can be included as *Range* elements subscribed to the */sonar_base* topic.

5.- Now add the laser *LaserScan* element and make it subscribe the topic */scan*. You will see the detected points on the rviz space.

6.- Add a *Camera* element and make it subscribe the */xtion/rgb/image_raw* topic.

7.- As both robots come with a depth camera, a *DepthCloud* element can be included. The depth map topic is */xtion/depth_registered/image_raw* and the color image topic is */xtion/rgb/image_raw*.

NOTE: The topic names for each robot can be different, buth the types are the same.

9.- Use the *key_teleop* demo to move around the robot and see how the data is displayed on rviz (see Fig. 5.3).

10.- Save the configuration to use it later when you implement your controllers. Save the file in the *config* folder of the *ics_gazebo* packages named as ROBOT.rviz. Remember to change ROBOT to *tiago* or *hsrb* depending on the robot you are using.

5.2 Using the default controllers and the controller manager

For now, this section must be executed using the TIAGo robot.

As described at the beginning of this section, when the TIAGo robot simulation is launched, a default set of controllers is loaded on the controller manager. These controllers listen to com-

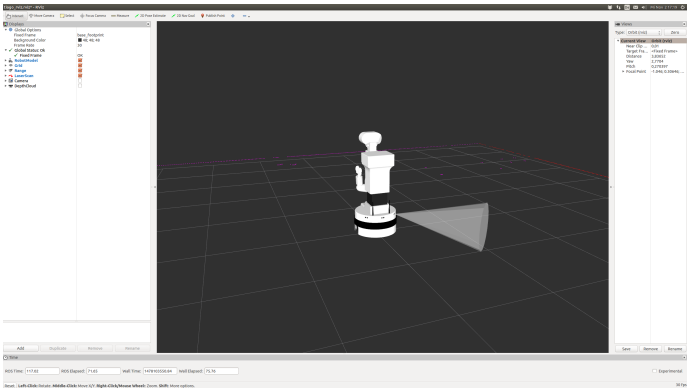

Figure 5.3: TIAGo robot displayed in rviz with all its sensors.

mand topics for desired positions (or velocities in the case of the mobile base controller) and drive those commands to the hardware interface with a PID control law. The next steps will help you to understand how the controller manager works.

1.- Launch a simulation of the TIAGo robot on the empty world.

2.- Run the next command in another terminal to list all the controllers currently running on the robot. There is a controller for every group of joints (torso, head, arm, gripper and base) and for the sensors.

```
$ rosservice call /controller_manager/list_controllers
```

3.- Publish a *geometry_msgs/Twist* message to the */mobile_base_controller/cmd_vel* topic using the *rostopic* node at a rate of three times per second. The next command will do it (You can find this command in the README.txt file from the template):

```
$ rostopic pub /mobile_base_controller /cmd_vel geometry_msgs/Twist "linear:
 x: 1.0
 y: 0.0
 z: 0.0
angular:
 x: 0.0
 y: 0.0
 z: 0.0" -r 3
```

4.- You will see the robot moving forward. Stop it by killing the process on the terminal where you launched the *rostopic* node.

5.- Now publish a command to the arm controller. Here you can specify a position in radians for every joint of the arm (You can find this command in the README.txt file from the template.).

```
$ rostopic pub /arm_controller/command trajectory_msgs/JointTrajectory "header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
joint_names: ['arm_1_joint', 'arm_2_joint', 'arm_3_joint', 'arm_4_joint', 'arm_5_joint', 'arm_6_joint', 'arm_7_joint']
points:
  -
    positions: [0.5, -1.34, -0.2, 1.94, -1.57, 1.37, 0.0]
    velocities: []
    accelerations: []
    effort: []
```

```
time_from_start:
  secs: 1
  nsecs: 0"
```

5.3 Create your own controller plugin

Other controllers can be created as plugin libraries if the default controllers are not enough to fulfil the application needs. A controller plugin can load any resource from the hardware interface and execute code on real time at the robot's loop rate. Controller plugins must be created in a ROS package following the next requirements.

1.- Locate from the provided material the *controllers_tutorials* folder. This folder is a ROS package with a basic controller structure to command the joints of the arm and read the IMU sensor. Open the folder and locate its components:

- **Source C++ file:** This is the most important file and contains the code for the controller. It must be a *cpp* file inside the *src* folder. The controller is created in a c++ class child of the controller_base class.
- **Config file:** A file containing information for the controller manager. This *yaml* file contains the controller name, and the resources used by the controller.
- **Launch file:** This file contains the configuration parameters to load the controller on the controller manager and the command to do it. This file is optional because the controller can also be loaded using a service call on the controller manager node.
- **Plugin xml file:** This file contains information for the controller manager to load it as a plugin on a certain namespace.
- **ROS package files:** The CMakeLists.txt file and the package.xml file that every ROS package must have.

2.- Open and read all these files to get familiar with the structure of the package. The features that must match among all the files are:

- **Controller name:** Stated inside the *yaml* file. It is recommended to name the yaml file, the launch file and the c++ source file with the same name as the controller. The *args* parameter inside the launch file must match this name too. For the provided example, this name is *combined_resource_controller*
- **Controller family namespace:** The namespace defined for a family of controllers. It is recommended to be the same as the ROS package. It is required that the namespace matches on the c++ source file, the *type* in the yaml file and the plugin xml file. For the provided example, the namespace is *controllers_tutorials*
- **Controller Class name:** The name of the class declared on the c++ source file. This class name must be the same in the type argument of the yaml file, the c++ source code and inside the plugin xml file. For the provided example the class name is *CombinedResourceController*

3.- To test a developed controller, it must be compiled in the catkin_workspace with all the other packages. Copy the *combined_resource_controller* folder inside the *tiago_tutorials* folder in the TIAGo workspace.

4.- Compile the workspace as shown in Section 2.1.3.

5.- Launch a simulation environment with the TIAGo robot on the empty world.

6.- In another terminal run the next command to load the new controller on the controller manager.

```
$ roslaunch controllers_tutorials combined_resource_controller.launch
```

7.- The control manager complained about a resource conflict because two controllers attempted to command the same actuator group. Therefore, in order to load the new controller, first the arm controller must be stopped. Stop the arm controller with the next command.

```
$ rosrun controller_manager controller_manager kill arm_controller
```

8.- Now try to launch the controller again. You will see the robot moving the arm in a sine-wave pattern.

9.- Stop the controller

```
$ rosrun controller_manager controller_manager kill combined_resource_controller
```

6 Exercises

The next subsections detail the exercises to be evaluated for this tutorial.

6.1 Create a simulation scenario

Using the tools provided in this tutorial, build a scenario (Gazebo world file) according to one of the robocup@home or World Robot Summit challenges. You **must** include the door model created in Section 4 and at least 20 objects including furniture and manipulable objects.

NOTE: Save the world file with the name *tutorial2.world* in the worlds folders of the *ics_gazebo* package. Also, make sure that the world is loaded when the following command is executed.

```
$ roslaunch ics_gazebo ROBOT.launch world_suffix:=tutorial2
```

6.2 Create an rviz configuration file

Follow the steps of Section 5.1 to read all the sensor data of one of the robots in rviz and save the configuration file in the *ics_gazebo* package. Save the file as *tiago.rviz* or *hsrb.rviz* depending on the robot you used.

NOTE: Make sure that the name of this file is correct and is loaded when the following command is executed.

```
$ roslaunch ics_gazebo ROBOT.launch
```

6.3 Default controllers

Adapt the position controller from the Tutorial 1 to control the mobile base of one of the robots. The HSRB robot has an omidirectional mobile base and TIAGo has a differential mobile base. The main difference is that for TIAGo, the mobile base is non holonomic. For this tutorial a controller for the Cartesian position of the frontal reference point of the robot must be implemented. The model of a differential mobile robot is showed in Fig. 6.1

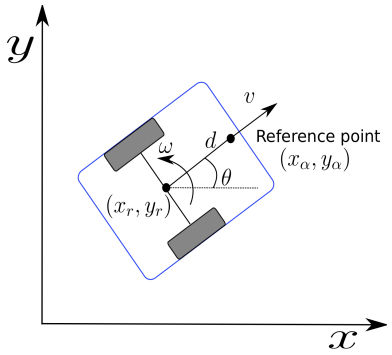

Figure 6.1: Kinematic model of a diferential robot.

To avoid singularities and implement a simple control law, the position to control will not be the point between the wheels but a reference point 30 centimetres in front of it. The dynamic of the reference point is described by:

$$\begin{bmatrix} \dot{x}_\alpha \\ \dot{y}_\alpha \end{bmatrix} = \begin{bmatrix} \cos\theta & -d\sin\theta \\ \sin\theta & d\cos\theta \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{6.1}$$

$$X = \begin{bmatrix} x_\alpha \\ y_\alpha \end{bmatrix} = \begin{bmatrix} x_r + d\cos\theta \\ y_r + d\sin\theta \end{bmatrix} \qquad T = \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{6.2}$$

Being X the Cartesian position of the reference point, T the command vector for linear speed v and rotational speed ω and θ the orientation of the robot in the world coordinate frame. Then, a simple control law to control the Cartesian position of the reference point is:

$$e = X_d - X \qquad \dot{X}_d = Ke \qquad K \in \mathbb{R}^2 \tag{6.3}$$

Where $\dot{X}_d$ is a desired Cartesian velocity command to send to the robot and K is a gain matrix. Finally to transform the Cartesian command into a twist command to send to the robot we can use the model equation as:

$$T = \begin{bmatrix} \cos\theta & -d\sin\theta \\ \sin\theta & d\cos\theta \end{bmatrix}^{-1} \dot{X}_d \tag{6.4}$$

Implement this simple control law on a ROS node using $d = 0.3$ and $K = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$

If you chose the HSRB robot, the controller design is the same as in Tutorial 1.

The topic to read the position of TIAGo is */mobile_base_controller/odom* with *nav_msgs/Odometry* messages and the topic and message to command the base are the same as in section 5.2.

The topic to read the position of HSRB is */hsrb/odom* with *nav_msgs/Odometry* messages and the topic to command the base is */base_velocity* with *geometry_msgs/Twist* messages.

As the orientation in the Odometry message is given in quaternions, you can use the next code to convert a quaternion to Euler angles in the Eigen library.

```
Quaterniond q;
q.x() = msg->pose.pose.orientation.x;
q.y() = msg->pose.pose.orientation.y;
q.z() = msg->pose.pose.orientation.z;
q.w() = msg->pose.pose.orientation.w;

Vector3d euler = q.toRotationMatrix().eulerAngles(0, 1, 2);
```

NOTE: You must deliver the package to run your program as well as the list of commands to run on terminals to execute it for the evaluation. Remember to explicitly include which robot you are using in the read me file.

6.4 Create a controller plugin

Based on the plugin structure described in Section 5.3, follow the next steps to create a controller plugin to control the torso joint.

1.- Create a new c++ source file inside the src folder in the *controllers_tutorials* package. Name it *new_torso_controller.cpp*

2.- Copy all the code from the *combined_resource_controller.cpp* file into the new file. Then, modify the class name to match the description in 5.3.

3.- Repeat steps 1 and 2 with the yaml files inside the config folder. The joint to control is named *torso_lift_joint*. Then make sure that the class name matches the c++ source file.

4.- Repeat steps 1 and 2 with the launch files. Again make sure that the namespace and the class name is properly set.

5.- Open the CMakeLists.txt of the package and add a line to compile the new source file. Follow the comments (start with # symbol) to include it properly.

6.- Open the *tutorial_controller_plugins.xml* file and follow the comments on it to source the new plugin. Make sure that all the names are correct.

7.- Compile the package and test the new controller following the steps of Section 5.3.

NOTE: For the evaluation of this exercise you will provide the modified *controllers_tutorials* package as well as the commands needed to run your solution.

7 Delivery

Create a read-me file containing four sections, one for every exercise. Start every section with the name of the robot you used in that section, followed with the name of the package (or

packages) required for every section. Then, write all the commands to compile, run and operate your solution. Compress the read-me file and the required packages to compile and run your solutions into a zip file and name it as:

Name_lastName_roboCupHome_tutorial2

Pay attention to the following remarks:

- **DO NOT** copy all the Workspace into the zip file. Copy only the packages needed for the solution.
- Be clear when writing your read me file.
- Specify which robot you used in the first line of every section.
- Include the list of the packages needed for every solution.
- Include all the commands to compile, execute and operate your solutions.
- Follow the naming conventions specified in this Document.
- Make sure that your solution compiles and runs in a single try.
- Deliver your solution before the deadline **5th of November at 23:59**.