ROS indigo and Gazebo2 Interface for the Pioneer3dx Simulation Ubuntu 14.04 LTS (Trusty Tahr)

Jen Jen Chung

February 22, 2016

Abstract

This document outlines the basic setup required to operate the Pioneer3dx simulation in Gazebo and ROS. The latter half of this document also provides a brief tutorial on the ROS publish/subscribe and rosservice message passing mechanisms in C++. These instructions are for an Ubuntu 14.04 LTS (Trusty Tahr) machine with the ROS indigo distribution, which interfaces with Gazebo2. Future ROS distributions may require different versions of Gazebo, users are encouraged to check online for the latest information at http://wiki.ros.org and http://gazebosim.org.

1 Installation

1.1 ROS Installation

The complete installation procedure and documentation for a **Desktop-Full Install** can be found at http://wiki.ros.org/indigo/Installation/Ubuntu. An abridged version of these instructions are given below.

In a new shell, copy and paste each of the following commands:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.
    list'

wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - | sudo apt-key add -

sudo apt-get update

sudo apt-get install ros-indigo-desktop-full

sudo rosdep init

rosdep update

echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc

source ~/.bashrc

sudo apt-get install python-rosinstall
```

If you only want to run and use ROS packages and do not wish to use Gazebo, go to Section 2.

1.2 Gazebo Installation

The installation instructions at http://gazebosim.org/tutorials?tut=install&ver=2.2&cat=install are for Ubuntu versions up to 13.10, however the setup for 14.04 is very similar and is provided below. Note that ROS indigo currently interfaces with Gazebo2, this may change in the future and new packages will be required.

In a new shell, copy and paste each of the following commands:

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu trusty main" > /etc/apt/sources.list.
    d/gazebo-latest.list'

wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
```

```
sudo apt-get update

sudo apt-get install gazebo2
```

When attempting to install `gazebo2`, apt-get may fail due to additional required dependencies (unlike in other cases, apt-get will not automatically include these in the install). You can proceed with the installation by listing all the package dependencies together with `gazebo2` in the install line.

Check the installation by typing in the command line:

```
gazebo
```

This will run two programs, `gzserver` and `gzclient`. The first time `gzserver` is called, it may take a few minutes to load all the Gazebo models. Once both programs are running, you will see a new Gazebo window with an empty world (ground plane model only). In the **Insert** tab on the left will be two expandable menus, one listed as **/.gazebo/models** and the second as **http://gazebosim.org/models**. The first is a local directory of Gazebo models and the second is its online model repository. You can optionally download these models to your local directory by visiting http://old.gazebosim.org/models/, downloading and unzipping the model.tar.gz files in each folder to ∼/.gazebo/models/.

To fully set up the Gazebo environment variables copy and paste the following commands into a shell:

```
export GAZEBO_MODEL_PATH=~/.gazebo/models:$GAZEBO_MODEL_PATH

export GAZEBO_RESOURCE_PATH=/usr/share/gazebo-2.2:$GAZEBO_MODEL_PATH

source ~/.bashrc
```

1.3 Install GAZEBO_ROS_PKGS

Documentation for these packages is provided at http://wiki.ros.org/gazebo_ros_pkgs. The ROS package **gazebo_ros_pkgs** includes three sub-packages: **gazebo_msgs**, **gazebo_plugins** and **gazebo_ros**, which are necessary to interface with the Gazebo simulator.

To install, open a new shell and copy and paste the following command:

```
sudo apt-get install ros-indigo-gazebo-ros-pkgs
```

Check the installation by typing in the command line:

```
roscore & rosrun gazebo_ros gazebo
```

This will open a new Gazebo window with an empty world. In a new shell, you can type:

```
rostopic list
```

to see a list of the available rostopics that are being published by Gazebo such as:

```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/rosout
/rosout_agg
```

To close Gazebo and end ROS, hold Ctrl+C in the shell from which you opened it and then type:

```
killall roscore
```

2 Simulating the Pioneer3dx

2.1 Set Up CATKIN Workspace

A good set of tutorials for how to set up a catkin workspace is given in http://wiki.ros.org/catkin/Tutorials. The pertinent parts of these tutorials are reproduced below.

The following commands will set up your catkin workspace in your home folder and source your shell environment. Be careful to change `user_name` to your user name in the echo command.

```
mkdir -p ~/catkin_ws/src

cd ~/catkin_ws/src

catkin_init_workspace

cd ~/catkin_ws

catkin_make

echo "source /home/user_name/catkin_ws/devel/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

To check that your workspace is properly overlaid by the setup script, enter the following command in a shell:

```
echo $ROS_PACKAGE_PATH
```

You should see the output:

```
/home/youruser/catkin_ws/src:/opt/ros/indigo/share:/opt/ros/indigo/stacks
```

2.2 Install Additional ROS Packages

Additional ROS packages (may not be included in the full desktop installation) are required for 2D navigation and robot control. These can be downloaded and installed via the command line:

```
sudo apt-get install ros-indigo-navigation ros-indigo-slam-gmapping ros-indigo-ros-control ros-indigo-ros-
    controllers ros-indigo-rviz
```

The **navigation** package provides waypoint navigation capabilities. It takes in information from odometry, sensor streams, and a goal pose to output safe velocity commands that are sent to a mobile base. Documentation is provided at http://wiki.ros.org/navigation.

The **slam_gmapping** package provides SLAM capabilities. It takes in laser and pose data to generate a 2D occupancy grid map of the environment. Documentation is provided at http://wiki.ros.org/slam_gmapping.

The **ros_control** package contains controller interfaces and allows the simulation of actuators in the robot model. Documentation is provided at http://wiki.ros.org/ros_control.

The **ros_controllers** package is a library of available controller plugins. It is used in conjunction with the **ros_control** package. Documentation is provided at http://wiki.ros.org/ros_controllers.

The **rviz** package is a 3D visualisation tool for ROS, it is able to display published rostopics of various message types and is often used to show sensor data and SLAM maps. URDF models can be interpreted by both Gazebo and RViz to aid visualisation. Documentation is provided at http://wiki.ros.org/rviz.

The following subsections describe how to download and run the Pioneer3dx simulator. For instructions on how to set up your own packages go to Section 3.

2.3 Download PIONEER_GAZEBO_ROS

The **pioneer_gazebo_ros** package is by no means solely my own code. Much of the code is adapted from the **p2os** package (available through apt-get and at https://github.com/allenh1/p2os), as well as from the **gazebo_ros_demos** package (see the tutorials at http://gazebosim.org/tutorials/?tut=ros_urdf and http://gazebosim.org/tutorials/?tut=ros_control, package available from https://github.com/ros-simulation/gazebo_ros_demos.git).

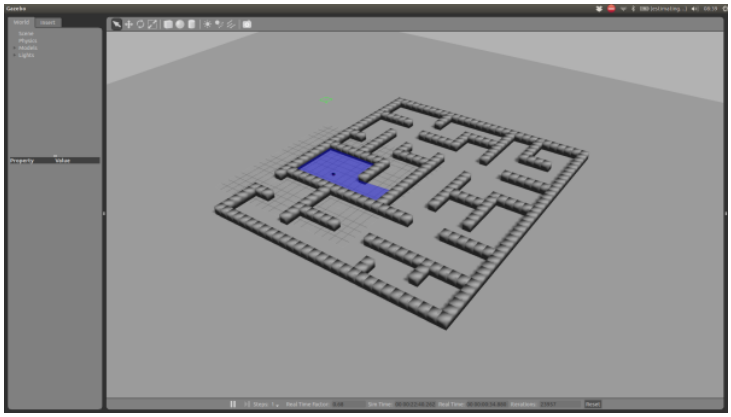

Figure 1: Gazebo static maze world including Pioneer3dx robot with Hokuyo laser sensor and forward-facing camera.

The **pioneer_2dnav** package contains the relevant .yaml configuration and launch files to run the navigation stack with the outputs of the **pioneer_gazebo_ros** package. In essence, this package should be able to run with any Pioneer3dx robot or robot simulator given the required odometry and laser sensor observations.

From the command line, navigate to your ROS workspace /src folder:

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

Download the packages by entering:

```
git clone https://github.com/JenJenChung/pioneer_gazebo_ros.git

git clone https://github.com/JenJenChung/pioneer_2dnav.git

git clone https://github.com/JenJenChung/nav_bundle.git

git clone https://github.com/JenJenChung/simple_navigation_goals.git
```

Make the packages:

```
cd ~/catkin_ws

catkin_make
```

You can now run the Pioneer3dx simulation by entering:

```
cd ~/catkin_ws/src/pioneer_gazebo_ros

./run\_pioneer\_gazebo
```

This will start an instance of **gazebo_ros** and **rviz** and launch the necessary files to perform 2D goal point navigation for a Pioneer3dx robot.

Note that occasionally, ROS or Gazebo may fail to run due to various runtime connection issues. Under these circumstances, hold Ctrl+C in the shell where the program was called and run `htop` to make sure all all Gazebo processes have terminated (SIGTERM any running instances of gzserver or gzclient). You can also run `killall roscore` to terminate a running roscore instance. After these programs have been terminated you can again attempt `./run_pioneer_gazebo`.

If launched successfully, you will see the pioneer simulation in the Gazebo window and the rostopic outputs in the RViz window as shown in Figures 1 and 2, respectively. In the RViz window, you can specify a 2D navigation waypoint by clicking on the **2D Nav Goal** button along the top bar and then selecting a location in the RViz map. Holding down the cursor will also allow you to adjust the goal point heading. A method for sending goal poses to the navigation stack via a ros node is shown in http://library.isr.ist.utl.pt/docs/roswiki/navigation%282f%29Tutorials%282f%29SendingSimpleGoals.html.

If a valid goal point is selected, that is, a collision-free path can be computed from the current robot pose to the goal, then the planned path will display as a green line in RViz and the robot will begin to move along this path. The pink line shows the current lookahead section of the path and the blue line shows the local path of the robot.

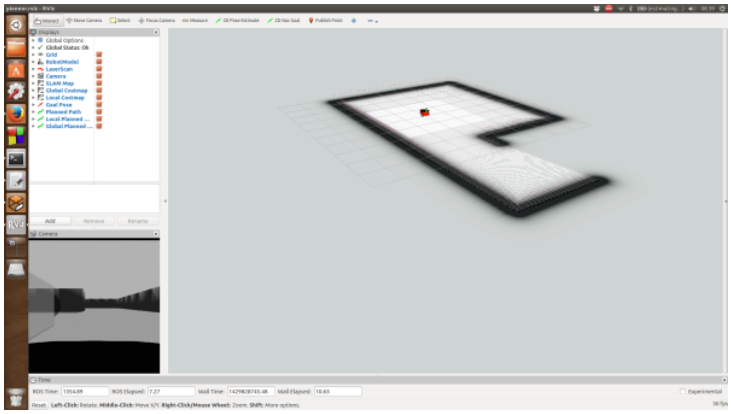

Figure 2: RViz world representation showing the Pioneer3dx robot with forward-facing camera and 360° laser sensor, as well as laser returns, SLAM map, global and local costmaps. The camera frame is shown in the bottom left inset.

Three maps are overlaid in RViz, the SLAM map, global costmap and local costmap. The two costmaps are used by the navigation stack to plan paths to the goal pose. The global costmap is essentially a blurred SLAM map that expands the boundaries of obstacles to accommodate the size and kinodynamics of the robot through user-defined parameters. The local costmap is used to perform a similar planning task with the corresponding local planner aiming to achieve the final lookahead pose along the global path given the locally observed obstacle constraints. If the robot strays beyond a threshold distance of the global path or the current global and local costmaps invalidate the current path, a new path will be planned from the current pose to the goal pose.

2.4 The Code Explained

There are three packages within **pioneer_gazebo_ros** that each contain launch files or publishers that are called by the main bash script `./run_pioneer_robot`.

2.4.1 PIONEER_DESCRIPTION

This package contains the URDF models (.xacro extensions) of the robot and sensors. URDF (universal robot description format) is an XML format for representing a robot model. It contains information regarding the physical components of the model (<links>) and how these components are connected (<joints>). Links have three main sets of properties: <visual>, <collision> and <inertial>. Visual properties define how the model will appear in Gazebo and RViz, collision properties define the physical space that the link occupies, and inertial properties define the mass and inertia of the link. Joints simply define the connection type between a parent and a child link as either fixed, revolute or continuous. The <origin> property specifies the location of the child link origin with respect to the parent link origin. For revolute and continuous joints. the axis property must be specified and defines the axis about which the child link can rotate. In addition, revolute joints must also specify limit efforts, joint damping and friction.

The main model is the pioneer3dx.xacro file and was adapted from the **p2os** package. Changes from the original package include setting the collision and inertial properties for each link, the addition of the camera and laser models as well as updating link1 to base_link to follow **navigation** naming conventions. Two transmissions were also included to act as the actuators at each wheel.

The URDF model is used to determine the transformations of each link as the robot moves. Moreover, it provides a transformation from the sensor frames to the odometry frame to enable SLAM mapping. To view the complete transform tree, you can use the command:

```
rosrun tf view_frames
```

This will produce a .pdf displaying the URDF link connections and any other transformations that are broadcast. See Figure 3 for an example of the transform tree for the Pioneer simulation.

Supplementary RViz model details are provided in pioneer3dx_wheel.xacro and materials.xacro, while pioneer.gazebo includes ROS plugin details for the Gazebo interface.

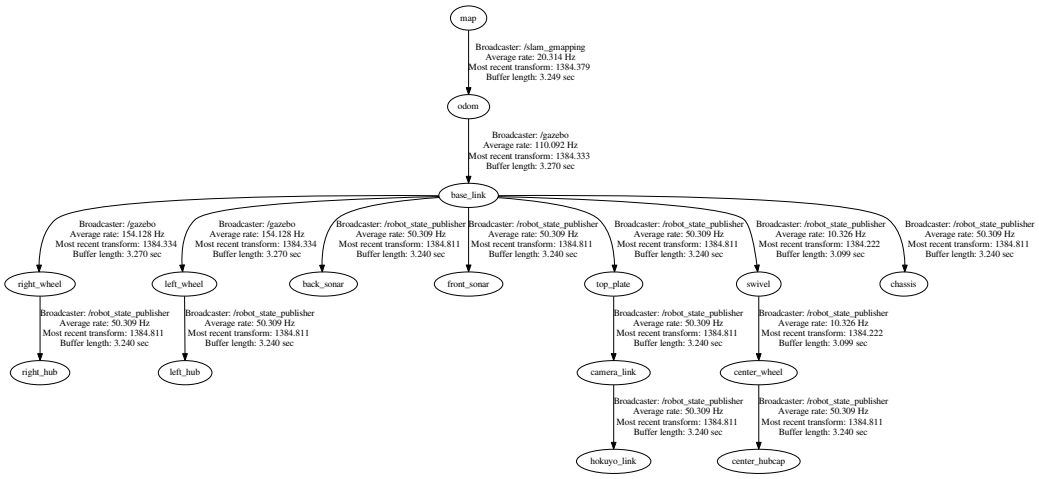

Figure 3: Complete transform tree for the Pioneer simulator.

2.4.2 PIONEER_GAZEBO

This package contains the .world models and .launch file to spawn the specified robot model and Gazebo world when running **gazebo_ros**. The Gazebo .world files are in XML format and existing models are easily added into the world by specifying the <uri>, e.g. model://model_name, within the <include> environment. The model_name should match a folder name inside ∼/.gazebo/models/ or a model name from the online model repository **http://gazebosim.org/models**. Note that requesting models from the online repository requires a solid internet connection and will take a minute or so to load, if you need things to load quickly, it is best to have a local copy in your ∼/.gazebo/models/ folder.

Changing the .world file to launch is simply a matter of changing the file name in pioneer_world.launch (line 12). This launch file will also spawn the robot model from the URDF file provided on line 22. Note that the two **find** commands on lines 12 and 22 are referring to the ROS packages (catkin folders) where it expects to find the specified files.

2.4.3 PIONEER_ROS

This package contains the C++ files to perform robot odometry and controller housekeeping. There are three .cpp files in the /src folder that are each required for the normal operation of the **slam_gmapping** package and the **move_base** controller. The pioneer_tf_broadcaster.cpp file requests the robot state from Gazebo through the rosservice /gazebo/get_model_state, which returns the robot pose and twist. A tf::TransformBroadcaster is then generated to broadcast the transformation from the robot base_link to the odometry frame. The pioneer_odom_publisher.cpp file works in conjunction with the broadcaster to perform the transformation and then publish the robot pose in the odometry frame. More information on transform broadcasters and rostopic publishers can be found in the next section.

The final .cpp file in this package, pioneer_pid_controller.cpp, is used to apply PID control to the /cmd_vel messages published by the /move_base package. At the moment, this code is only set up to run as a P controller; the integral and derivative gains are not used to tune the commanded velocities. The program reads in the PID gains from the pioneer_controller_params.yaml and applies these values through a callback function, which is triggered whenever a new /cmd_vel message is received. The tuned velocity commands are published under a new topic name, /pioneer/cmd_vel; this topic name must match the <commandTopic> specified in the Gazebo controller plugin definition in pioneer.gazebo (inside **pioneer_gazebo**). Note that the <robotNamespace> property can be used to prefix all following topics, this may be useful when simulating multiple robots.

To link the .cpp files such that they are compiled in a `catkin_make` call, the program details must be included in the CMakeLists.txt document inside the package folder. Any changes to the .cpp files will require a call to `catkin_make` from the catkin folder to compile the new code.

2.4.4 PIONEER_2DNAV

The **pioneer_2dnav** package contains the relevant .yaml configuration files and a .launch file to load these parameters and run the **move_base** node in the navigation stack. Typically, to run the navigation stack for 2D waypoint path planning, you will need a package similar to this one that contains parameter values relevant to your robot. The parameters in the four configuration files are described below. More information on the available parameters and how to tune them can be found at http://wiki.ros.org/navigation/Tutorials/RobotSetup.

base_local_planner_params.yaml

This file contains parameters related to the robot kinematics. Minimum and maximum command velocities are defined here as well as robot acceleration limits. Most properties are self-explanatory; a few less-obvious parameters are:

- escape_vel---the velocity to apply when robot is stuck (should be negative so that robot reverses)
- sim_time---amount to time to forward-simulate trajectories in seconds
- meter_scoring---boolean to denote if distance measurements are in metres
- heading_lookahead---distance in metres along in-place rotation trajectories to compare headings
- pdist_scale---weight describing importance of robot sticking close to path
- gdist_scale---weight describing importance of robot reaching goal

The pdist_scale and gdist_scale seem to operate on a relative manner, a much higher gdist_scale will cause the robot to cut corners more aggressively, while a higher pdist_scale will encourage the robot to closely track the planned path. See http://wiki.ros.org/base_local_planner for more information about the adjustable parameters.

costmap_common_params.yaml

As suggested by the file name, the parameters set in this .yaml file are applied to both the local and global planners. The robot footprint and the sensor parameters would typically be included in this file. It is also acceptable to not include a costmap_common_params.yaml file if you choose to define the required parameters separately in the local and global parameter files. The opposite (i.e. only use the common configuration file) is not advisable since the local and global planners perform quite different planning tasks and often need different to operate at different resolutions. See http://wiki.ros.org/costmap_2d for more information.

global_costmap_params.yaml

The global costmap update and publish frequencies can afford to be lower than that of the local planner since it does not need to account for local collision avoidance. The inflation radius determines the obstacle buffer size (either in cells or metres depending on the meter_scoring parameter). The buffer zone contains a varying cost field and exists outwards from the region of collision (region occupied by obstacle + robot footprint). The shape of the cost field, that is, how quickly the cost decays to 0, is determined by the cost_scaling_factor. The larger this value, the faster the cost decays. Setting the value too high causes the planner to suggest paths that traverse very close to regions of high cost and this can result in jerky execution. If noisy sensors or a less-agile robot is used, map updates may frequently find the robot in unexpected high cost regions, forcing aggressive correction manoeuvres.

local_costmap_params.yaml

The local costmap update and publish frequencies should be as high or higher than that of the global costmap to ensure active local collision avoidance. The static_map parameter should be set to false since the local costmap should move with the robot as it traverses the space. Setting the rolling_window parameter to true will allow the planner to discard old maps in the world, which are no longer necessary for local collision avoidance (the map information will still be stored in the global costmap).

The width and height of the local costmap should be judiciously chosen, taking into account the robot sensor range and the environment that it will be operating in. The local planner assigns a path that attempts to match the final global path heading that is visible within the local map window. Thus, if the local window is set too large, the local planner will attempt turns much earlier and, depending on the environment, will be more prone to getting stuck. See an extreme case in Figure 4.

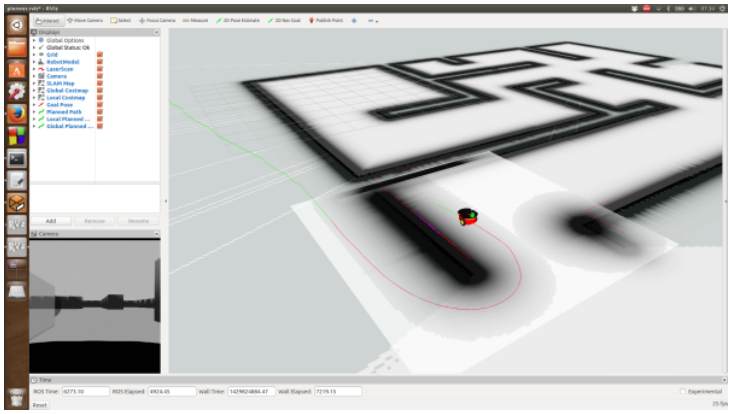

Figure 4: The robot's local costmap frame is too large and expands across to the other side of a wall that it has not yet observed. The global path traverses along one side of the wall and in the opposite direction on the other side, because of this, the robot local planner in its current position is attempting to match the pose at the end of the pink line, which requires the exact opposite heading to that which will allow it to progress along the path. These two competing objectives means that the commanded velocity goes to 0 and the robot is stuck.

In terms of the local cost_scaling_factor, experience suggests that setting the local cost_scaling_factor lower than the global cost_scaling_factor results in smoother trajectories. A low local cost scaling allows for smoother low level collision avoidance manoeuvres when new obstacles are observed, such as when turning around corners. The difference between the local and global cost scaling can be seen in Figure 4. In the highlighted square around the robot (local costmap), the obstacles have a larger grey buffer region than elsewhere in the field (global costmap).

3 Creating a ROS Package

An excellent set of tutorials can be found at http://wiki.ros.org/ROS/Tutorials. The following section derives partially from the turtlesim example that is described in these tutorials with a focus on the ROS framework required to set up 2D navigation for a moving robot.

To create a new package in your catkin workspace, navigate to the ~/catkin_ws/src folder in a shell and type:

```
catkin_create_pkg pkg_name pkg_dep1 pkg_dep2 pkg_depn
```

Where `pkg_name` is replaced by the name of your package and `pkg_dep1`, etc. are replaced with the package dependencies. Package dependencies are defined as the set of other packages that must exist for this package to compile and operate. Defining them here sets up a compile-time check in CMakeLists.txt to ensure that the package can run.

If successfully created, a new folder will exist inside the /src folder under the name of your package and will include:

CMakeLists.txt **include** package.xml **src**

The package dependencies will be listed in the CMakeLists.txt file as `catkin REQUIRED COMPONENTS` and in package.xml using `<build_depend>` and `<run_depend>` tags. You can add additional dependencies by including them in these two files using the same format. Any source code for your package should be stored in the /src folder while launch files should be stored in a separate /launch folder (`mkdir launch`) within the package directory.

The following subsection describes how to create your own robot model for use in ROS and Gazebo, if you are using an existing robot ROS package such as RosAria you can skip to Section 3.2 to learn how to write your own ROS subscribers and publishers

3.1 Robot Model

Many commercial-off-the-shelf robots will have a ROS interface package that handles the robot model transformations, such as RosAria for Adept MobileRobots and TurtleBot for iRobot. However, if a model does not exist, then you will need to write your own URDF model. A series of ROS tutorials for how to write URDF models can be found at http://wiki.ros.org/urdf/Tutorials. A brief summary of the various components is provided below.

3.1.1 URDF Models

URDF (universal robot description format) is an XML format for representing a robot model. It contains information regarding the physical components of the model (<links>) and how these components are connected (<joints>). The URDF model can also define the available actuation through <transmission> blocks.

Links have three main sets of properties: <visual>, <collision> and <inertial>. Visual properties define how the model will appear in Gazebo and RViz, collision properties define the physical space that the link occupies, and inertial properties define the mass and inertia of the link. Typically you would set the visual and collision geometries to be the same and both support the use of .stl and .dae mesh files (CAD models).

Joints define the connection type between a parent and a child link as either fixed, revolute or continuous. The <origin> property specifies the location of the child link origin with respect to the parent link origin. For revolute and continuous joints. the axis property must be specified and defines the axis about which the child link can rotate. In addition, revolute joints must also specify limit efforts, joint damping and friction.

As a general rule of thumb, it is easiest to set the link origin to **xyz="0 0 0" rpy="0 0 0"** (in the visual, collision and inertial properties), and define the translation and rotation of the link through the joint connection. Furthermore, when using this model with the navigation stack, convention requires the definition of a "base_link" from which to describe the odometry frame. The base_link can be a physical link of the robot or an empty link that simply defines the reference origin for all other links in the model, as implemented in the Pioneer3dx model and shown in Figure 3.

To enable actuation, <transmission> properties must be included in the model. The transmission block defines the actuator properties at a specified joint and sets up the interface for various Gazebo controller plugins. A simple example of a two-wheeled robot with a two actuation joints and a camera is given below in Listing 1.

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

    <link name="base_link"/>

    <joint name="chassis_joint" type="fixed">
        <origin xyz="-0.045 0 0.148" rpy="0 0 0"/>
        <parent link="base_link"/>
        <child link="chassis"/>
    </joint>

    <link name="chassis">
        <visual name="chassis_visual">
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <geometry>
                <box size="0.43 0.277 0.17"/>
            </geometry>
            <material name="ChassisRed">
                <color rgba="0.851 0.0 0.0 1.0"/>
            </material>
        </visual>
        <collision>
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <geometry>
                <box size="0.43 0.277 0.17"/>
            </geometry>
        </collision>
        <inertial>
            <mass value="5.67"/>
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <inertia ixx="0.07" ixy="0" ixz="0"
```

```
            iyy="0.08" iyz="0" izz="0.10"/>
    </inertial>
</link>

<joint name="base_left_wheel_joint" type="continuous">
    <origin xyz="0 0.155 0.093" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
    <parent link="base_link"/>
    <child link="left_wheel"/>
</joint>

<link name="left_wheel">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <cylinder radius="0.092" length="0.04"/>
        </geometry>
        <material name="WheelBlack">
            <color rgba="0.117 0.117 0.117 1.0"/>
        </material>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <cylinder radius="0.092" length="0.04"/>
        </geometry>
    </collision>
    <inertial>
        <mass value="0.1"/>
        <origin xyz="0 0 0"/>
        <inertia ixx="0.012411765597" ixy="-0.000711733678" ixz="0.00050272983"
            iyy="0.015218160428" iyz="-0.000004273467" izz="0.011763977943"/>
    </inertial>
</link>

<joint name="base_right_wheel_joint" type="continuous">
    <origin xyz="0 -0.155 0.093" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
    <parent link="base_link"/>
    <child link="right_wheel"/>
</joint>

<link name="right_wheel">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <cylinder radius="0.092" length="0.04"/>
        </geometry>
        <material name="WheelBlack">
            <color rgba="0.117 0.117 0.117 1.0"/>
        </material>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <cylinder radius="0.092" length="0.04"/>
        </geometry>
    </collision>
    <inertial>
        <mass value="0.1"/>
        <origin xyz="0 0 0"/>
        <inertia ixx="0.012411765597" ixy="-0.000711733678" ixz="0.00050272983"
            iyy="0.015218160428" iyz="-0.000004273467" izz="0.011763977943"/>
    </inertial>
</link>

<joint name="camera_joint" type="fixed">
    <origin xyz="0.125 0 0.2595" rpy="0 0 0"/>
    <parent link="base_link"/>
    <child link="camera_link"/>
</joint>

<link name="camera_link">
```

```
        <visual>
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <geometry>
                <box size="0.05 0.07 0.10"/>
            </geometry>
            <material name="CameraGreen">
                <color rgba="0.0 0.8 0.0 1.0"/>
            </material>
        </visual>
        <collision>
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <geometry>
                <box size="0.05 0.07 0.10"/>
            </geometry>
        </collision>
        <inertial>
            <mass value="1e-5" />
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
        </inertial>
    </link>

    <transmission name="tran1" type="transmission_interface/SimpleTransmission">
        <joint name="base_left_wheel_joint">
            <hardwareInterface>EffortJointInterface</hardwareInterface>
        </joint>
        <actuator name="motor1">
            <hardwareInterface>EffortJointInterface</hardwareInterface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>

    <transmission name="tran2" type="transmission_interface/SimpleTransmission">
        <joint name="base_right_wheel_joint">
            <hardwareInterface>EffortJointInterface</hardwareInterface>
        </joint>
        <actuator name="motor2">
            <hardwareInterface>EffortJointInterface</hardwareInterface>
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>

</robot>
```

Listing 1: URDF model for a two-wheeled robot

3.1.2 Gazebo Plugins

Gazebo plugins in ROS are the typical method for managing controllers in the Gazebo simulation, from driving the robot to operating the sensors. Desired plugins are defined within the .gazebo file, which contains all the Gazebo-specific model properties such as the appearance of the model in Gazebo and the sensor parameters for simulating observations. This file should be included at the start of the robot URDF model to be loaded together at runtime.

Standard sensor plugins exist for a variety of common sensors (laser, RGB/D camera, etc.) and require the name of the sensor link from the URDF model for observation frame to odometry frame transformation purposes. Other sensor parameters may also be specified in the plugin definition and vary from sensor to sensor. An example of an RBG camera Gazebo description and plugin is provided in Listing 2.

```
<?xml version="1.0"?>
<robot>
    <gazebo reference="camera_link">
        <sensor type="camera" name="camera1">
            <update_rate>30.0</update_rate>
            <camera name="head">
                <horizontal_fov>1.3962634</horizontal_fov>
                <image>
                    <width>800</width>
                    <height>800</height>
                    <format>R8G8B8</format>
```

```
            </image>
            <clip>
                <near>0.02</near>
                <far>300</far>
            </clip>
            <noise type="gaussian">
                <mean>0.0</mean>
                <stddev>0.007</stddev>
            </noise>
        </camera>
        <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
            <robotNamespace>/</robotNamespace>
            <alwaysOn>true</alwaysOn>
            <updateRate>0.0</updateRate>
            <cameraName>pioneer/camera1</cameraName>
            <imageTopicName>image_raw</imageTopicName>
            <cameraInfoTopicName>camera_info</cameraInfoTopicName>
            <frameName>camera_link</frameName>
            <hackBaseline>0.07</hackBaseline>
            <distortionK1>0.0</distortionK1>
            <distortionK2>0.0</distortionK2>
            <distortionK3>0.0</distortionK3>
            <distortionT1>0.0</distortionT1>
            <distortionT2>0.0</distortionT2>
        </plugin>
    </sensor>
  </gazebo>
</robot>
```

Listing 2: RGB camera Gazebo URDF model and plugin definition

Different driver plugins also exist for a variety of robot configurations (two-wheeled differential drive, four-wheeled skid steer, etc.). Typically these plugins require access to the transmission joints (joint name from URDF model) and the rostopics that define the robot odometry frame and the **move_base** commanded velocity. An example of the differential drive controller plugin for the two-wheeled robot in Listing 1 is given in Listing 3 below.

```
<?xml version="1.0"?>
<robot>
  <gazebo>
    <plugin filename="libgazebo_ros_diff_drive.so" name="differential_drive_controller">
        <robotNamespace>/</robotNamespace>
        <alwaysOn>true</alwaysOn>
        <updateRate>100</updateRate>
        <leftJoint>base_left_wheel_joint</leftJoint>
        <rightJoint>base_right_wheel_joint</rightJoint>
        <torque>5</torque>
        <wheelSeparation>0.39</wheelSeparation>
        <wheelDiameter>0.15</wheelDiameter>
        <commandTopic>pioneer/cmd_vel</commandTopic>
        <odometryTopic>odom</odometryTopic>
        <odometryFrame>odom</odometryFrame>
        <robotBaseFrame>base_link</robotBaseFrame>
        <publishWheelTF>true</publishWheelTF>
        <publishWheelJointState>true</publishWheelJointState>
        <wheelAcceleration>0</wheelAcceleration>
        <wheelTorque>5</wheelTorque>
        <rosDebugLevel>na</rosDebugLevel>
    </plugin>
  </gazebo>
</robot>
```

Listing 3: Differential drive controller plugin

See http://gazebosim.org/tutorials?tut=ros_gzplugins for more details on the available Gazebo plugins and their parameters.

3.2 Subscribers and Publishers

There are a number of ways to access and modify rostopic information. From the command line, you can identify the available rostopics by typing:

```
rostopic list
```

Provided roscore and RosAria are running, this may produce an output such as:

```
/RosAria/battery_recharge_state
/RosAria/battery_state_of_charge
/RosAria/battery_voltage
/RosAria/bumper_state
/RosAria/cmd_vel
/RosAria/motors_state
/RosAria/parameter_descriptions
/RosAria/parameter_updates
/RosAria/pose
/RosAria/sonar
/RosAria/sonar_pointcloud2
/odom
/rosout
/rosout_agg
/tf
```

To output the value of any topic, you can request an echo to the shell:

```
rostopic echo -n1 /RosAria/cmd_vel
```

which will output the next values that are published for that topic, for example,

```
linear:
  x: 0.25
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

The `-n1` term in the echo command requests a single output, if you wish to display all published values as they arrive, simply remove this component. Other echo options are also available, type `rostopic echo --help` in the command line for more information.

To publish to a rostopic, whether through the command line or via an external program, you must send the information as the correct message type. You can identify the message type through the command line using:

```
rostopic type /RosAria/cmd_vel
```

which in this case will output:

```
geometry_msgs/Twist
```

As seen in the previous output, twist messages include 3-axis elements representing the commanded linear velocities as well as the commanded angular velocities. You can verify the individual components of this message type using:

```
rosmsg show geometry_msgs/Twist
```

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Publishing a new /RosAria/cmd_vel message from the command line can be done via a number of formats:

```
rostopic pub -1 /RosAria/cmd_vel geometry_msgs/Twist -- '[1.0, 0.0, 0.0]' '[0.0, 0.0, 0.5]'

rostopic pub -1 /RosAria/cmd_vel geometry_msgs/Twist '{linear: {x: 1.0, y: 0.0, z: 0.0}, angular: {x: 0.0,
     y: 0.0, z: 0.5}}'
```

Both commands will send the same message. Note that the second format is consistent with publishing for all message types where nesting is indicated using {} and message types are identified by name followed by a colon and a space. The `-1` option means that the message will publish once and then exit, see `rostopic pub --help` for information on other options.

3.2.1 Writing Your Own Subscriber/Publisher Node

To subscribe and publish to existing or new topics within your own package can be achieved in a number of ways, the following will describe the methods that have been implemented within the **pioneer_gazebo_ros** package. See the ROS tutorial at http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29 for more information.

To subscribe to a rostopic, you can write a callback function that will be run whenever a new instance of the particular rostopic is received. For example, say we want to apply gains of 0.5 to the commanded velocities and then republish the new commands under a new topic name; this can be achieved via the following node:

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"

ros::Publisher pub ;

void cmd_velCallback(const geometry_msgs::Twist& msg)
{
    geometry_msgs::Twist current_cmd_vel = msg ;
    geometry_msgs::Twist new_cmd_vel ;

    // Tune commanded velocities
    new_cmd_vel.linear.x = current_cmd_vel.linear.x*0.5 ;
    new_cmd_vel.linear.y = current_cmd_vel.linear.y*0.5 ;
    new_cmd_vel.linear.z = current_cmd_vel.linear.z*0.5 ;
    new_cmd_vel.angular.x = current_cmd_vel.angular.x*0.5 ;
    new_cmd_vel.angular.y = current_cmd_vel.angular.y*0.5 ;
    new_cmd_vel.angular.z = current_cmd_vel.angular.z*0.5 ;

    // Publish new velocities
    pub.publish(new_cmd_vel) ;
}

int main(int argc, char** argv)
{
    // Initialise node gain_controller
    ros::init(argc, argv, "gain_controller");

    // Create the publisher, topic /tuned_cmd_vel with queue size 10
    ros::NodeHandle nHandpub ;
    pub = nHandpub.advertise<geometry_msgs::Twist>("/tuned_cmd_vel", 10) ;

    // Create the subscriber, topic /RosAria/cmd_vel with queue size 10
    // to request callback function cmd_velCallback
    ros::NodeHandle nHandsub ;
    ros::Subscriber sub = nHandsub.subscribe("/RosAria/cmd_vel", 10, &cmd_velCallback) ;
```

```
    // Return control to ROS
    ros::spin() ;
    return 0;
}
```

Listing 4: ROS package node **gain_controller**, which uses a callback function to tune and republish commanded velocities

In the main program, the ROS node **gain_controller** is initialised and ROS node handles are used to access the ROS subscriber and publisher functions. The publisher advertises to a rostopic called /tuned_cmd_vel, which may or may not already exist. If it does not exist, a new rostopic will be created. Be careful when advertising to existing rostopics, if there is more than one source publishing to that topic, the value of that topic will vary depending on the timing of the various source publications. The subscriber requests the /RosAria/cmd_vel rostopic and calls cmd_velCallback whenever a new message is received. In this example, the subscriber can buffer up to 10 messages if the callback function takes longer to execute than the message arrival rate. The callback function performs the necessary operations to the subscribed message and latches the new commanded velocities to the publisher. The final command, ros::spin(), returns control to ROS.

Note the header requirements in the node; all ROS message types must be imported using #include. Some message types are nested within others, for example, the nav_msgs/Odometry header also contains the header for geometry_msgs/Twist message types. Check the online message type documentation for more information (links provided in Section 4.2).

If the rostopic to be published does not require input from an existing rostopic, then a callback function is not needed. In this case, the publisher can simply be written within the main program as shown in Listing 5. This method uses a while loop whose settings determine the rate at which the rostopic messages are published.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"

int main(int argc, char** argv)
{
    // Initialise node cmd_controller
    ros::init(argc, argv, "cmd_controller");

    // Create the publisher, topic /new_cmd_vel with queue size 10
    ros::NodeHandle nHandpub ;
    ros::Publisher pub = nHandpub.advertise<geometry_msgs::Twist>("/new_cmd_vel", 10) ;

    // Publish at 10Hz
    ros::Rate loop_rate(10) ;

    while (ros::ok())
    {
        geometry_msgs::Twist new_cmd_vel ;

        // Define command velocities
        new_cmd_vel.linear.x = 1.0 ;
        new_cmd_vel.linear.y = 0.0 ;
        new_cmd_vel.linear.z = 0.0 ;
        new_cmd_vel.angular.x = 0.0 ;
        new_cmd_vel.angular.y = 0.0 ;
        new_cmd_vel.angular.z = 0.5 ;

        // Publish new velocities
        pub.publish(new_cmd_vel) ;

        // Return control to ROS for specified loop rate
        ros::spinOnce() ;
        loop_rate.sleep() ;
    }
    return 0;
}
```

Listing 5: ROS package node **cmd_controller**, which publishes commanded velocities

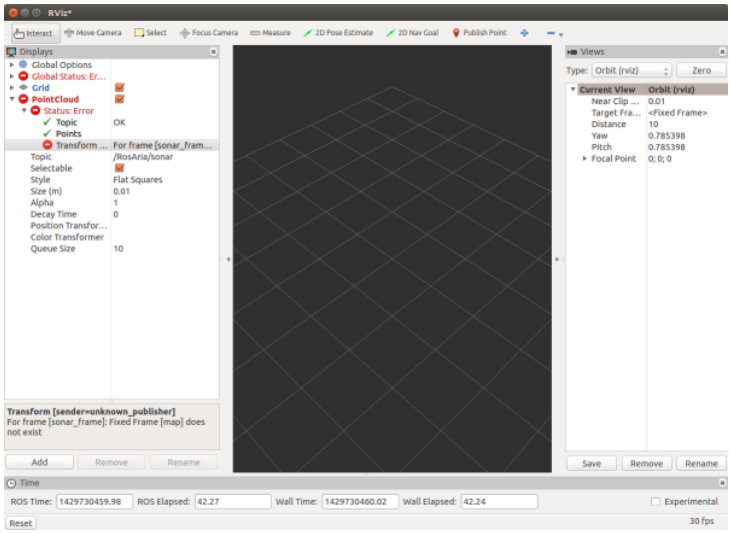

Figure 5: RViz transform frame error.

3.2.2 Linking and Compiling Nodes

Any time you create new code in your package /src folder, you must identify it in the package CMakeLists.txt file in order to compile and make the code. For example, say that you have written a ROS node called cmd_controller in controller.cpp for your ROS package **robot_controller**, the following lines must be in the **robot_controller** package CMakeLists.txt to correctly make the node.

```
cmake_minimum_required(VERSION 2.8.3)
project(robot_controller)

find_package(catkin REQUIRED COMPONENTS
  geometry_msgs
  roscpp
  rospy
  std_msgs
)

catkin_package()

include_directories(
  ${catkin_INCLUDE_DIRS}
)

add_executable(cmd_controller src/controller.cpp)
target_link_libraries(cmd_controller ${catkin_LIBRARIES})
add_dependencies(cmd_controller ${catkin_EXPORTED_TARGETS})
```

Listing 6: Adding C++ executables to the CMakeLists.txt file

Additional C++ files can be included by repeating the last three lines of Listing 6 and changing the node and file name. Calls to `catkin_make` will compile all source code listed in the CMakeLists.txt files of all available catkin packages.

3.3 Managing Frames of Reference

When using sensors or performing SLAM on the robot, transform frame management is critical. If you have set up your robot URDF model correctly, then this model will take care of the transformations from the sensors to the robot base_link. An easy way to test this is to run RViz and visualise the rostopic data from the sensors. The sensor returns will display if the model transformations exist, otherwise an error will appear in the left menu stating that the transform frame does not exist, as shown in Figure 5. If the error is in the model, you can check for breaks in the transformations by inspecting the transform tree using `rosrun tf view_frames` and add joints in the model where appropriate.

If you do not have a model, or you wish to define the transformation between the robot odometry frame and the global map for mapping, then the general methodology is to write a transform frame

broadcaster and a transform frame publisher. The broadcaster defines the transformation between the two listed reference frames and the frame publisher (listener) uses the broadcast transformation to advertise the data in the transformed reference frame. An example of a broadcaster node that defines the transformation between the odometry frame and the robot base_link is provided in Listing 7.

```
#include "ros/ros.h"
#include "nav_msgs/Odometry.h"
#include "tf/transform_broadcaster.h"

void poseCallback(const nav_msgs::Odometry& msg)
{
  static tf::TransformBroadcaster br ;

  tf::Transform transform ;
  geometry_msgs::Point pp = msg.pose.pose.position ;
  geometry_msgs::Quaternion qq = msg.pose.pose.orientation ;
  transform.setOrigin( tf::Vector3(pp.x, pp.y, pp.z) ) ;
  transform.setRotation( tf::Quaternion(qq.x, qq.y, qq.z, qq.w) ) ;
  br.sendTransform( tf::StampedTransform(transform, ros::Time::now(), "odom", "base_link") );
}

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

  ros::NodeHandle nHandle;
  ros::Subscriber sub = nHandle.subscribe("/RosAria/pose", 10, &poseCallback);

  ros::spin();
  return 0;
}
```

Listing 7: Example of a transformation frame broadcaster

In this example, the two frames are set to coincide with one another. A translation in the frames is indicated through the input to the setOrigin class function, while the input to setRotation represents a rotation between the frames. The corresponding transform publisher is shown in Listing 8.

```
#include "ros/ros.h"
#include "nav_msgs/Odometry.h"
#include "tf/transform_broadcaster.h"

ros::Publisher pub;

void poseCallback(const nav_msgs::Odometry& msg)
{
  geometry_msgs::Point pp = msg.pose.pose.position ;
  geometry_msgs::Quaternion qq = msg.pose.pose.orientation ;
  tf::Quaternion tf_quat(qq.x, qq.y, qq.z, qq.w) ;

  geometry_msgs::Twist current_Twist = msg.twist.twist ;

  geometry_msgs::TransformStamped odom_trans;

  odom_trans.transform.translation.x = pp.x ;
  odom_trans.transform.translation.y = pp.y ;
  odom_trans.transform.translation.z = pp.z ;

  geometry_msgs::Quaternion geo_Quat ;
  tf::quaternionTFToMsg(tf_quat, geo_Quat) ;
  odom_trans.transform.rotation = geo_Quat ;

  odom_trans.header.stamp = ros::Time::now() ;
  odom_trans.header.frame_id = "odom" ;
  odom_trans.child_frame_id = "base_link" ;

  nav_msgs::Odometry odom ;
  odom.header.stamp = odom_trans.header.stamp ;
  odom.header.frame_id = "odom" ;
  odom.child_frame_id = "base_link";

  // set the position
  odom.pose.pose.position.x = odom_trans.transform.translation.x ;
```

```
  odom.pose.pose.position.y = odom_trans.transform.translation.y ;
  odom.pose.pose.position.z = odom_trans.transform.translation.z ;
  odom.pose.pose.orientation = geo_Quat ;

  // set the velocity
  odom.twist.twist.linear.x = current_Twist.linear.x ;
  odom.twist.twist.linear.y = current_Twist.linear.y ;
  odom.twist.twist.linear.z = current_Twist.linear.z ;

  odom.twist.twist.angular.x= current_Twist.angular.x ;
  odom.twist.twist.angular.y= current_Twist.angular.y ;
  odom.twist.twist.angular.z= current_Twist.angular.z ;

  // publish the message
  pub.publish(odom);
}

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

  ros::NodeHandle nHandlePub ;
  ros::NodeHandle nHandleSub ;

  pub = nHandlePub.advertise<nav_msgs::Odometry>("odom", 50) ;
  ros::Subscriber sub = nHandleSub.subscribe("/RosAria/pose", 10, &poseCallback) ;

  ros::spin() ;
  return 0 ;
}
```

Listing 8: Example of an odometry publisher

3.4 ROS Services

Aside from subscribing and publishing to rostopics, another way to access data is via rosservices. To see the available rosservices from the command line, type:

```
rosservice list
```

Gazebo provides a number of rosservices through which to access simulation data, these typically exist in Get/Set pairs that allow users to read and write to various rostopics, respectively. The following example will demonstrate how to use rosservices to perform the transform broadcast from the odometry frame to the robot base_link, as in the previous section, for a robot simulated in Gazebo.

Listing 9 makes use of the /gazebo/get_model_state rosservice to access the current robot pose and orientation. Rosservices typically have request and response components, request components are used to identify a particular data stream if multiple are available, and response components return the requested data. In this example, there may be a number of models within the Gazebo simulation, the model_name and relative_entity_name inputs are used to identify the particular model requested by the rosservice call. The response components return the pose and orientation of the requested model to variables in the workspace.

```
#include "ros/ros.h"
#include "gazebo_msgs/GetModelState.h"
#include "nav_msgs/Odometry.h"
#include "tf/transform_broadcaster.h"

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

    // Initialise node and service client
    ros::NodeHandle n ;
    ros::ServiceClient client ;

    // Publish at 10Hz
    ros::Rate loop_rate(10);

    // GetModelState request variables
```

```
    std::string modelName = (std::string)"pioneer" ;
    std::string relativeEntityName = (std::string)"world" ;
    gazebo_msgs::GetModelState getModelState ;

    geometry_msgs::Point pp ;
    geometry_msgs::Quaternion qq ;

    static tf::TransformBroadcaster br ;
    tf::Transform transform ;

    while (ros::ok())
    {
        // Request robot state
        client = n.serviceClient<gazebo_msgs::GetModelState>("/gazebo/get_model_state") ;
        getModelState.request.model_name = modelName ;
        getModelState.request.relative_entity_name = relativeEntityName ;
        client.call(getModelState) ;

        pp = getModelState.response.pose.position ;
        qq = getModelState.response.pose.orientation ;

        // Compute and broadcast transformation
        transform.setOrigin( tf::Vector3(pp.x, pp.y, pp.z) ) ;
        transform.setRotation( tf::Quaternion(qq.x, qq.y, qq.z, qq.w) ) ;
        br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "odom", "base_link")) ;

        ros::spinOnce();
        loop_rate.sleep() ;
   }
   return 0;
}
```

Listing 9: Example of a transformation frame broadcaster using a rosservice to access the model state

The corresponding publisher can be created using the same /gazebo/get_model_state rosservice by combining the rosservice call syntax of Listing 9 with the transformation algorithm in Listing 8.

The request and response message types for each available Gazebo rosservice can be found by navigating through the message type documentation. A link is provided in Section 4.2.

3.5 Running Your Package

The convention for running multiple nodes from the same package is to write a .launch file, which should be included in a launch folder inside your package directory. The launch file lists all the ROS nodes to launch and any initial parameters required by each node. An example is shown in Listing 10

```
<launch>
  <node name="my_node1" pkg="my_pkg" type="my_executable1">
    <param name="param1" type="bool" value="true"/>
    <param name="param2" type="int" value="2"/>
    <param name="paramn" type="string" value="n"/>
  </node>

  <node name="my_node2" pkg="my_pkg" type="my_executable2"/>
</launch>
```

Listing 10: ROS launch file my_pkg.launch

This launch file can be run from the command line using:

```
roslaunch my_pkg my_pkg.launch
```

This will execute two ROS nodes, my_node1 and my_node2 from the ROS package my_pkg, it will search for these two nodes in the executables listed under the respective type property (note that C++ files do not use the file extension, whereas the .py extension is required for python scripts). The first node initialises with three parameters while the second does not access any initial parameters. Launch files can run nodes from multiple ROS packages, it is common to access inbuilt ROS packages in this way. More information regarding launch files can be found at http://wiki.ros.org/roslaunch/XML/node.

Sometimes it may be desirable to run individual nodes through the command line, especially when node execution is order-dependent. A simple way to do this without needing to `rosrun` each individual

node in a new shell is to combine the necessary ROS commands into a single bash script that will execute all the components in the correct order. Using this method you can also introduce timings and delays to allow programs time to startup before running other nodes that depend on its output signals. The bash script used to initialise the full Pioneer3dx Gazebo and ROS simulation is provided in Listing 11 as an example. Note that it is possible to create a single launch file that is equivalent to the bash script shown below, however, roslaunch has a bad habit of launching files out of order and causing crashes due to unmet dependencies. The interested reader is directed to the ROS rocon package for single masters, which can alleviate these problems within the ROS framework. In practice, however, launching through a bash script with well-timed sleeps has been demonstrated to provide sufficient robustness.

```
#!/bin/bash

my_pid=$$
echo "My process ID is $my_pid"

echo "Launching roscore..."
roscore &
pid=$!
sleep 5s

echo "Launching Gazebo..."
roslaunch pioneer_gazebo pioneer_world.launch &
pid="$pid $!"

sleep 5s

echo "Launching transform publishers..."
roslaunch pioneer_ros pioneer_ros.launch &
pid="$pid $!"

sleep 3s

echo "Launching navigation stack..."
roslaunch nav_bundle nav_bundle.launch &
pid="$pid $!"

sleep 3s

echo "Launching controller..."
roslaunch pioneer_ros pioneer_controller.launch &
pid="$pid $!"

echo "Launching Rviz..."
roslaunch pioneer_description pioneer_rviz.launch &
pid="$pid $!"

trap "echo Killing all processes.; kill -2 TERM $pid; exit" SIGINT SIGTERM

sleep 24h
```

Listing 11: Bash script for launching all components of the Pioneer3dx Gazebo and ROS simulator

This bash script can use both `roslaunch` and `rosrun` to execute launch files as well as individual nodes, respectively. The final `trap` call kills all processes started within this bash script when Ctrl+C is held in the executing shell.

4 Useful Links

4.1 Tutorials

Message type	URL
All ROS tutorials	http://wiki.ros.org/ROS/Tutorials
All Gazebo tutorials	http://gazebosim.org/tutorials

4.2 ROS Message Type Documentation

Message type	URL
gazebo_msgs	http://docs.ros.org/indigo/api/gazebo_msgs/html/index-msg.html
geometry_msgs	http://docs.ros.org/indigo/api/geometry_msgs/html/index-msg.html
nav_msgs	http://docs.ros.org/indigo/api/nav_msgs/html/index-msg.html
std_msgs	http://docs.ros.org/indigo/api/std_msgs/html/index-msg.html
tf	http://docs.ros.org/indigo/api/tf/html/index-msg.html