Lab exercise Nr. 1

This exercise will cover creating of robot model using URDF and running differential drive controller to issue velocity commands to this robot.

1. Create a package with the following name:
`simplebot_description`
This package will store the robot URDF model and all associated files. The package should contain all the necessary files (package.xml and CMakeLists.txt). <robot_name>_description is the standard naming convention for the package that contains robot description files.

2. Build and source the workspace where this package is located. Add source command to .bashrc file (optional). This will source the workspace automatically each time a new terminal is started.

3. Create `urdf` folder in `simplebot_description` package and `simplebot.urdf.xacro` file.

4. Now adjust `simplebot.urdf.xacro`. Add main robot links and joints. You can read more on this in URDF tutorials. The urdf files can also be simplified using Xacro meta-programming language and we are going to use it here. You can read more about xacro here. To create a realistic and accurate robot model you should add `<inertial>`, `<collision>` and `<visual>` tags to each link. Now your urdf should look like this:

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

  <!-- The definition of colors used in this urdf -->
  <material name="white">
    <color rgba="1 1 1 1"/>
  </material>
  <material name="black">
    <color rgba="0 0 0 1"/>
  </material>

  <!-- Property values used in this model -->
  <xacro:property name="M_PI" value="3.1415926535897931"/>

  <!--
  This is the main link of the robot. usually it is chosen to
be the centre
  point of the robot.
```

```
  -->
  <link name="base_link">
    <!-- Inertia is used to simulate the physics of the robot
in simulator -->
    <inertial>
      <mass value="1.0"/>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <!--
      This is the most tricly part. It can be calculated
using a mesh and
      programs such as MeshLab. If you get these values
totally wrong your robot
      might just explode of fly away in the simulator.
      -->
      <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001"
iyz="0" izz="0.001"/>
    </inertial>
    <!--
    Collisions are used in simulation as well. They help to
determine when the
    robot starts touching some other surface. This requires
some heavy maths so
    collision objects should be as simple as possible.
Ideally they should only
    be primitive shapes such as box, cylinder and sphere.
    -->
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <box size="0.7 0.2 0.075"/>
      </geometry>
    </collision>
    <!--
    This is only used for visualization purposes. This can be
a 3D CAD model of
    your robot.
  -->
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <box size="0.7 0.2 0.075"/>
      </geometry>
      <material name="white"/>
    </visual>
  </link>

  <!--
  Macro defining robot wheel. This macro can be used to
```

```
initialize multiple
  wheels. Macros can have parameters that make it possible to
change the behaviour
  of the actual URDF xml that is going to be generated when
this macro is instantiated
  -->
  <xacro:macro name="wheel" params="side side_ind">
    <!--
    Each new robot part should have its own link. These links
have the same
    fields as the base_link link.
    -->
    <link name="${side}_wheel">
      <inertial>
        <mass value="0.5"/>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <inertia ixx="0.001" ixy="0" ixz="0" iyy="0.001"
iyz="0" izz="0.001"/>
      </inertial>
      <collision>
        <origin xyz="0 0 0" rpy="${M_PI/2} 0 0"/>
        <geometry>
          <cylinder radius="0.075" length="0.05"/>
        </geometry>
      </collision>
      <visual>
        <origin xyz="0 0 0" rpy="${M_PI/2} 0 0"/>
        <geometry>
          <cylinder radius="0.075" length="0.05"/>
        </geometry>
        <material name="black"/>
      </visual>
    </link>
    <!--
    Joints are used to connect links together. There are
several type of joints
    that we can have. The most common are revolute and
continuous. You can read
    more about them in URDF documentation.
    -->
    <joint name="${side}_wheel_joint" type="continuous">
      <parent link="base_link"/>
      <child link="${side}_wheel"/>
      <xacro:if value="${side_ind}">
        <origin xyz="0.35 0.125 0" rpy="0 0 0"/>
      </xacro:if>
      <xacro:if value="${not side_ind}">
        <origin xyz="0.35 -0.125 0" rpy="0 0 0"/>
```

```
      </xacro:if>
      <axis xyz="0 1 0"/>
    </joint>
    <!--
    Transmissions are used by ros_control. They are necessary
so that we can
    simulate joints that are then controlled by ros_control
controllers
    -->
    <transmission name="${side}_wheel_tran">
      <!--
      Type of transmission. These can be more sophisticated.
Have a look at
      documentation to get more info on this.
      -->
      <type>transmission_interface/SimpleTransmission</type>
      <!--
      This should be the same as the joint that you defined
above. Again joints
      can be controlled in several ways. For example, we can
have velocity,
      position or effort controlled joints. This depends on
the motor controller
      that is used in the robot.
      -->
      <joint name="${side}_wheel_joint">

<hardwareInterface>hardware_interface/VelocityJointInterface<
/hardwareInterface>
      </joint>
      <!--
      Actuator is the same as the actual motor that rotates
the joint. The gear
      ration can be specified in mechanicalReduction
parameter.
      -->
      <actuator name="${side}_wheel_motor">
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>
  </xacro:macro>
  <!--
  Here you actually declare the wheels. This is the
initialization of the macro
  that we declared above. We also give different parameters
for each wheel
  -->
  <xacro:wheel side="left" side_ind="1"/>
```

```
  <xacro:wheel side="right" side_ind="0"/>

</robot>
```

Note. There is usually a problem when copying the following xml to a file. After copying fix the syntax in line 135!

5. Now create `launch` folder in your package. Here we will create the launch file `view_urdf.launch` that will visualize our shiny new robot in `rviz` program. This file should look like this:

```
<launch>
  <!--
  We first convert our robot_description from xacro to urdf.
This urdf is then
  loaded into ros parameter robot_description.
  -->
  <param name="robot_description" command="$(find
xacro)/xacro --inorder '$(find
simplebot_description)/urdf/simplebot.urdf.xacro'" />

  <!--
  We now start three nodes.
  joint_state_publisher is a package that is needed only when
you do not have
  the real robot. This node publishes joint_states topic.
This topic is needed so
  that robot_state_publisher can publish states of all joints
to TF.
  robot_state_publisher is used to publish robot joint states
to TF. All the static
  joints are published as is and moving joint angles are
taken from joint_states
  topic that is provided by joint_state_publisher. NOTE that
joint_state_publisher
  is only needed when we do not have the real (simulated)
robot.
  rviz is ros 3D tool used to visualize various robot sensor
data.
  -->
  <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="rviz" pkg="rviz" type="rviz" args="-d $(find
simplebot_description)/config/view_urdf.rviz" required="true"
/>
</launch>
```

After rviz is loaded you should first change the robot fixed frame to `base_link`. Then press add button and visualize Robot model and TF. You can then save this rviz config so that you do not have to do this each time you start rviz. Save the config to `config` folder in your `simplebot_description` package. Call it `view_urdf.rviz`. Notice that this file is loaded automatically in your launch file that you created in previous step.

6. Now let's try to load this robot model into Gazebo simulator. Create the following new launch file in your package `simplebot_gazebo.launch`:

```
<launch>

  <!-- Load the environment for simulation -->
  <!-- Here you can give a different world_name to load
different environments -->
  <include file="$(find
gazebo_ros)/launch/empty_world.launch">
    <!--
    NOTE: Uncomment this to launch a different world
    <arg name="world_name" value="$(find
simplebot_gazebo)/worlds/YOUR_WORLD.world""/>
    -->
    <arg name="paused" value="false"/>
  </include>

  <!-- The URDF of the robot -->
  <param name="robot_description"
           command="$(find xacro)/xacro --inorder
                   '$(find
simplebot_description)/urdf/simplebot.urdf.xacro'" />

  <!-- Spawn your robot in Gazebo simulator -->
  <node pkg="gazebo_ros" type="spawn_model"
name="spawn_model" args="-urdf -x 0 -y 0 -z 0.075 -param
robot_description -model SimpleBot"/>

</launch>
```

You can try to right click on your robot and enable different views of your robot physical properties. You can also try pushing your robot around by applying some forces to it.

7. Now we would actually like to move our robot around. We will use ros_control and diff_drive_controller to help us do this. This will require several modifications. First we need to add ros_control plugin to robot description file `simplebot.urdf.xacro`:

```
  <gazebo>
    <plugin name="gazebo_ros_control"
```

```
filename="libgazebo_ros_control.so" />
  </gazebo>
```

This will simulate the HardwareInterface of your robot that is needed to run robot controllers. Now lets create a separate package for our robot control related stuff. Call this package `simplebot_control`. Create the necessary files for this package. Create `launch` and `config` folders in this new package. Add the following to `simplebot_controllers.launch` place this file in `launch` folder:

```
<launch>
  <!-- Run the robot simulation file that we previously
created -->
  <include file="$(find
simplebot_description)/launch/simplebot_gazebo.launch"/>

  <!-- Load controller configurations from YAML file to
parameter server -->
  <rosparam file="$(find
simplebot_control)/config/simplebot_controllers.yaml"
command="load"/>

  <!--
  Here we only need robot_state_publisher because
joint_states are going to be
  published by our created controller
  -->
  <node name="robot_state_publisher"
pkg="robot_state_publisher" type="robot_state_publisher" />

  <!--
  This node loads all robot controllers. The names of the
controllers should
  match the names you give to each controller in the yaml
file.
  -->
  <node name="controller_spawner" pkg="controller_manager"
type="spawner" respawn="false" args="joint_state_controller
simplebot_controller"/>
</launch>
```

Now add simplebot_controllers.yaml to the config folder with the following content:

```
# We will load two controllers here.
#
# First controller (joint_state_controller) takes the joint
states from
# Gazebo simulator and publishes them to /joint_states topic.
This information
# is used to determine the position of each joint at any
```

```
given point in time.
# When we viewed URDF before joint_state_publisher node was
used to publish these
# states but they were static and did not change over time.
Now we want to get the
# actual joint states from simulated robot. In a real robot
you would have to
# implement RobotHardwareInterface class that reads these
states from real robot.
#
# Second controller (simplebot_controller) is a
diff_drive_controller. It takes
# as an input a Twist message. This message contains the
velocity of robot center
# point around various axis (x, y, theta). It than converts
this information into
# a velocity for each of your robot wheel. In addition it
takes the state of each
# of your robots joints and calculates the odometry of your
robot.

# this first line is the name of the controller. It can be
arbitrary and you decide
# how to call your controllers.
joint_state_controller:
  # this defines the actual type of the loaded controller.
  type: joint_state_controller/JointStateController
  # the following are all the parameters of your controller.
You can read about
  # the parameters that your controller has in its
documentation page
  publish_rate  : 80

# our second robot controller
simplebot_controller:
  # its type is a diff_drive_controller
  type: diff_drive_controller/DiffDriveController
  # now we define the remaining parameters.
  left_wheel: 'left_wheel_joint'
  right_wheel: 'right_wheel_joint'
  pose_covariance_diagonal: [0.001, 0.001, 1000000.0,
1000000.0, 1000000.0, 1000.0]
  twist_covariance_diagonal: [0.001, 0.001, 1000000.0,
1000000.0, 1000000.0, 1000.0]
```

Note that this file also has some copying problems. Make sure that all comments are single line. And parameters of each controller are indented two spaces. You will also have to install several additional packages. Run the following command in terminal:

```
sudo apt install ros-kinetic-ros-control
sudo apt install ros-kinetic-ros-controllers
sudo apt install ros-kinetic-gazebo-ros-control
```

Now we can start your robot with some control capabilities. Try launching your robots control launch file `simplebot_controllers.launch`.

8. Let's inspect what new things we can now do as our robot now has a controller. First open `rviz` and load the same environment configuration you created before. You should see your robot but also a new transform called `odom`. Also add a `odom` topic visualization and change the fixed frame from `base_link` to `odom`. Later when you will have your robot moving try changing the fixed frame from one frame to the other and see what that does to your robot.

9. You would probably like to move your robot around. Open `rqt` and start `Robot Steering` plugin. This plugin makes it possible for you to move your robot around. At the top of the opened window change the topic that is used to publish robot Twist messages. This should be changed to `/simplebot_controller/cmd_vel`. Note that `rqt` has some other useful tools such as a Plot tool. Try using plot tool to visualize robot `/joint_states` as the robot is moving. Play around with different tools and see how they work.

That's about it for this exercise. For the L1 defense you will be asked to make some small changes to these files. First there are things that you all have to add before L1 defence:

1. Add a third free wheel to your robot so that is is easier for your robot to move. Alternatively you can add two wheels at the back and make your robot a skid steer robot. Look at the diff_drive controller documentation on how to configure skid steering. Freewheel will not require any additional controller changes.
2. Add distance sensors to your robot. You can either add ultrasonic ranging sensors or a lidar. You will use this to avoid obstacles in P1. For laser have a look here. For sonar look here. If you want you can use bumper or any other sensor.
3. Add a wall to your gazebo world so that the robot can follow it.

The things I might ask to do during defence:

1. Add a new joint to your robot.
2. Modify parameters of the robot. Make wheels bigger etc.
3. Visualize different robot joint states in Plot.

4. For fun. Your SimpleBot is quite lonely. Why not add a dragon on top of your robot? Find a free dragon stl and add it as an additional visual to your robot. This is not mandatory.
5. For adventurous only! Add additional controller for the new joint. This can either be a velocity_controller or position_controller. Note that documentation is quite poor on this and you might have to dig a bit in source. You are more than welcome to improve documentation of ROS :).

If you get stuck somewhere feel free to e-mail me and ask questions!

Note for extremely lazy students! All the code for this exercise is available here.