The Construct
Learn and develop for robots

FULL PRACTICE ROS PROJECT MANUAL

EXPLORING ROS WITH WHEELED ROBOTS

LEARN & TEACH ROS WITH NO HASSLE

WWW.THECONSTRUCTSIM.COM

Written by **MARCO ANTONIO ARRUDA**

Edited by **VIVEK YOGI** and **YUHONG LIN**

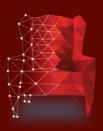
Cover design by **YUHONG LIN**

The Construct

Learn and develop for robots with ROS

About

This project is about using a Two Wheeled Mobile Robot to explore features and tools provided by ROS (Robot Operating System). We start building the robot from the scratch, using URDF (Unified Robot Description Format) and RViz to visualize it. Further, we describe the inertia and show how to simplify the URDF using XACROS. Later, motion planning algorithms, such as Obstacle Avoidance and Bugs 0, 1 and 2 are developed to be used in the built robot. Some ROS packages, like robot_localization, are used to built a map and localize on it.



The Construct

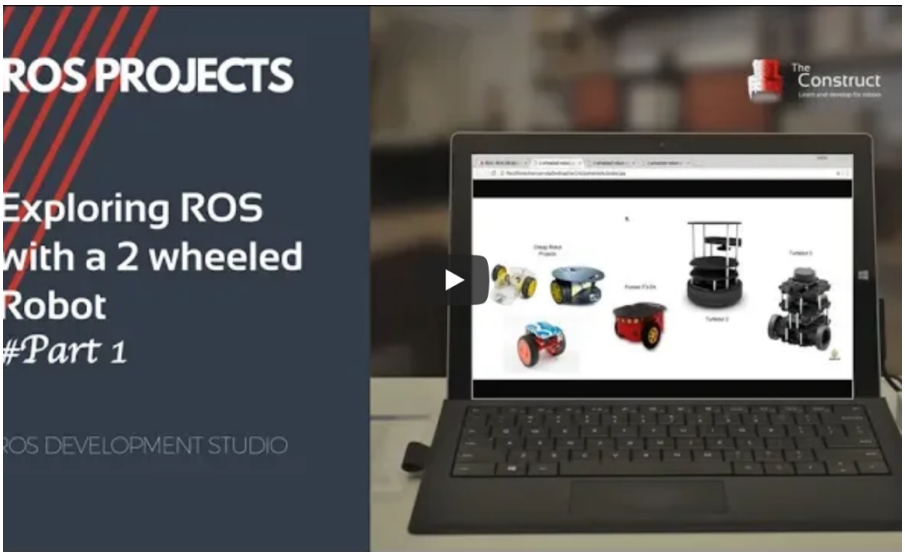
Learn and develop for robots with ROS

INDEX

Exploring ROS with a 2 Wheeled Robot #Part 1

In this video, we are going to explore the basics of robot modeling using the Unified Robot Description Format (URDF). At the end of this video, we will have a model ready and running in Gazebo simulator.

Video Tutorial Part 1



Step 1.1

- Head to [ROS Development Studio] and create a new project.
- Provide a suitable **project name** and some useful **description**.
- Open the project (this will take few seconds)
- Once the project is loaded run the ***IDE*** from the tools menu. Also verify that the initial **directory structure** should look like following:

```
.
├── ai_ws
├── catkin_ws
│   ├── build
│   ├── devel
│   └
├── notebook_ws
│   ├── default.ipynb
│   └── images
└── simulation_ws
    ├── build
    ├── devel
    └── src
```

The directory simulation_ws is also called **simulation workspace**, it is supposed to contain the code and scripts relevant for simulation. For all other files we have the catkin_ws (or catkin workspace). A few more terminologies to familiarize are xacro and macro, basically xacro is a file format encoded in xml. xacro files come with extra features called macros (akin functions) that helps in reducing the amount of written text in writing robot description. Robot model description for Gazebo simulation is described in URDF model format and xacro files simplify the process of writing elaborate robot description.

Step 1.2

Now we will create a catkin package with name **m2wr_description**. We will add rospy as dependency.
Start a **SHELL** from tools menu and navigate to **~simulation_ws/src** directory as follows

```
$ cd simulation_ws/src
```

To create catkin package, use the following command

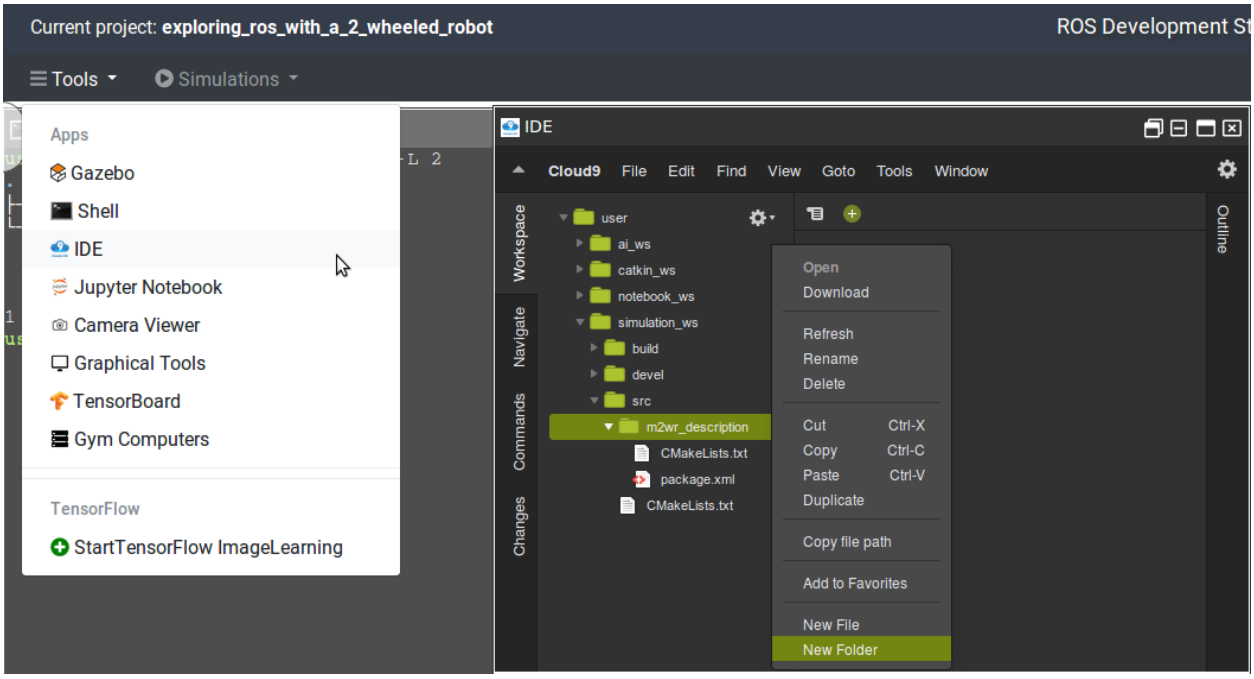
```
$ catkin_create_pkg m2wr_description rospy
```

At this point we should have the following **directory structure**

```
.
├── ai_ws
├── catkin_ws
│   ├── build
│   ├── devel
│   └── 
├── notebook_ws
│   ├── default.ipynb
│   └── images
└── simulation_ws
    ├── build
    ├── devel
    └── src
        ├── CMakeLists.txt
        └── m2wr_description
            └── CMakeLists.txt
```

```
17          └── package.xml
```

Create a directory named **urdf** inside **m2wr_description** directory. Create a file named **m2wr.xacro** inside the newly created **urdf** directory. Creating files and directories is easier (right mouse click and select appropriate option) using the **IDE** from the **Tools** menu option.



We will populate the xacro file with the following content

```
1  <?xml version="1.0" ?>
2
3  <robot name="m2wr" xmlns:xacro="https://www.ros.org/wiki/xacro" >
4
5    <material name="black">
6      <color rgba="0.0 0.0 0.0 1.0"/>
7    </material>
8    <material name="blue">
9      <color rgba="0.203125 0.23828125 0.28515625 1.0"/>
10   </material>
11   <material name="green">
12     <color rgba="0.0 0.8 0.0 1.0"/>
13   </material>
14   <material name="grey">
15     <color rgba="0.2 0.2 0.2 1.0"/>
16   </material>
17   <material name="orange">
18     <color rgba="1.0 0.423529411765 0.0392156862745 1.0"/>
19   </material>
20   <material name="brown">
21     <color rgba="0.870588235294 0.811764705882 0.764705882353 1.0"/>
22   </material>
```

```
23  <material name="red">
24    <color rgba="0.80078125 0.12890625 0.1328125 1.0"/>
25  </material>
26  <material name="white">
27    <color rgba="1.0 1.0 1.0 1.0"/>
28  </material>
29
30  <gazebo reference="link_chassis">
31    <material>Gazebo/Orange</material>
32  </gazebo>
33  <gazebo reference="link_left_wheel">
34    <material>Gazebo/Blue</material>
35  </gazebo>
36  <gazebo reference="link_right_wheel">
37    <material>Gazebo/Blue</material>
38  </gazebo>
39
40  <link name="link_chassis">
41    <!-- pose and inertial -->
42    <pose>0 0 0.1 0 0 0</pose>
43
44    <inertial>
45      <mass value="5"/>
46      <origin rpy="0 0 0" xyz="0 0 0.1"/>
47      <inertia ixx="0.0395416666667" ixy="0" ixz="0" iyy="0.106208333333" iyz="0" izz="0.106208333333"/>
48    </inertial>
49
50    <collision name="collision_chassis">
51      <geometry>
52        <box size="0.5 0.3 0.07"/>
53      </geometry>
54    </collision>
55
56    <visual>
57      <origin rpy="0 0 0" xyz="0 0 0"/>
58      <geometry>
59        <box size="0.5 0.3 0.07"/>
60      </geometry>
61      <material name="blue"/>
62    </visual>
63
64    <!-- caster front -->
65    <collision name="caster_front_collision">
66      <origin rpy=" 0 0 0" xyz="0.35 0 -0.05"/>
67      <geometry>
68        <sphere radius="0.05"/>
69      </geometry>
70      <surface>
71        <friction>
72          <ode>
73            <mu>0</mu>
74            <mu2>0</mu2>
75            <slip1>1.0</slip1>
76            <slip2>1.0</slip2>
77          </ode>
78        </friction>
79      </surface>
80    </collision>
81    <visual name="caster_front_visual">
82      <origin rpy=" 0 0 0" xyz="0.2 0 -0.05"/>
83      <geometry>
84        <sphere radius="0.05"/>
```

```
85      </geometry>
86    </visual>
87    </link>
88
89  <!-- Create wheel right -->
90
91  <link name="link_right_wheel">
92    <inertial>
93      <mass value="0.2"/>
94      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
95      <inertia ixx="0.00052666666" ixy="0" ixz="0" iyy="0.00052666666" iyz="0" izz="0.001"/>
96    </inertial>
97
98    <collision name="link_right_wheel_collision">
99      <origin rpy="0 1.5707 1.5707" xyz="0 0 0" />
100     <geometry>
101       <cylinder length="0.04" radius="0.1"/>
102     </geometry>
103   </collision>
104
105   <visual name="link_right_wheel_visual">
106     <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
107     <geometry>
108       <cylinder length="0.04" radius="0.1"/>
109     </geometry>
110   </visual>
111
112 </link>
113
114 <!--   Joint for right wheel  -->
115 <joint name="joint_right_wheel" type="continuous">
116   <origin rpy="0 0 0" xyz="-0.05 0.15 0"/>
117   <child link="link_right_wheel" />
118   <parent link="link_chassis"/>
119   <axis rpy="0 0 0" xyz="0 1 0"/>
120   <limit effort="10000" velocity="1000"/>
121   <joint_properties damping="1.0" friction="1.0" />
122 </joint>
123
124 <!-- Left Wheel link -->
125
126 <link name="link_left  wheel">
127   <inertial>
128     <mass value="0.2"/>
129     <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
130     <inertia ixx="0.00052666666" ixy="0" ixz="0" iyy="0.00052666666" iyz="0" izz="0.001"/>
131   </inertial>
132
133   <collision name="link_left_wheel_collision">
134     <origin rpy="0 1.5707 1.5707" xyz="0 0 0" />
135     <geometry>
136       <cylinder length="0.04" radius="0.1"/>
137     </geometry>
138   </collision>
139
140   <visual name="link_left_wheel_visual">
141     <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
142     <geometry>
143       <cylinder length="0.04" radius="0.1"/>
144     </geometry>
145   </visual>
146
```

```
147   </link>
148
149   <!--   Joint for right wheel  -->
150   <joint name="joint_left_wheel" type="continuous">
151     <origin rpy="0 0 0" xyz="-0.05 -0.15 0"/>
152     <child link="link_left_wheel" />
153     <parent link="link_chassis"/>
154     <axis rpy="0 0 0" xyz="0 1 0"/>
155     <limit effort="10000" velocity="1000"/>
156     <joint_properties damping="1.0" friction="1.0" />
157   </joint>
158
159 </robot>
```

In tis xacro file we have defined the following:

- **chasis + caster wheel**: link type element
- **wheels**: link type elements (left + right)
- **joints**: joint type elements (left + right)

All of these elements have some common properties like inertial, collision and visual. The inertial and collision properties enable physics simulation and the visual property controls the appearance of the robot.

The joints help in defining the relative motion between links such as the motion of wheel with respect to the chassis. The **wheels** are links of cylindrical geometry, they are oriented (using rpy property) such that rolling is possible. The placement is controlled via the xyz property. For joints, we have specified the values for damping and friction as well. Now we can visualize the robot with rviz.

Step 1.3

To visualize the robot we just defined, we will create a lunch file named **rviz.launch** inside **launch** folder. We will populate it with following content:

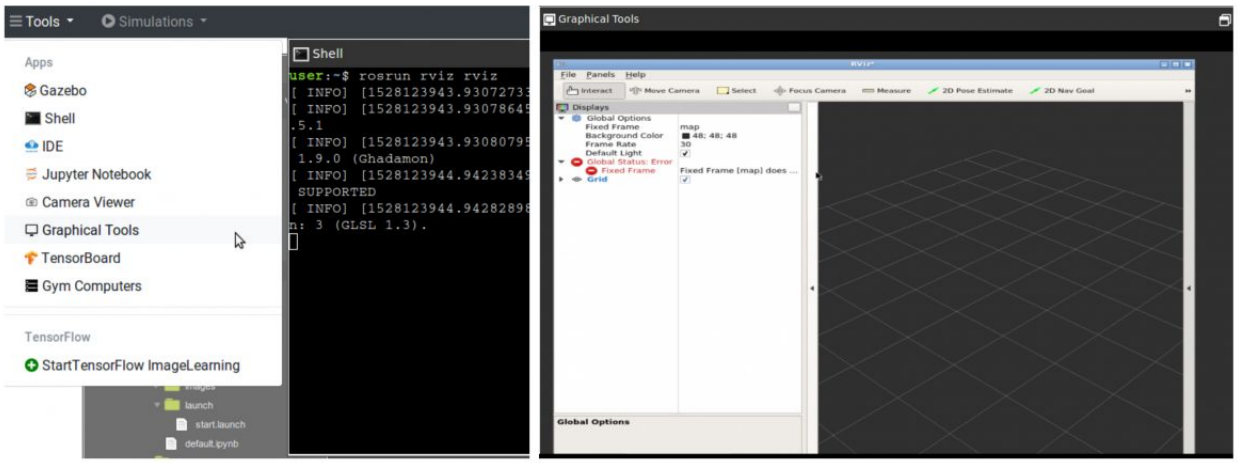
```
1  <?xml version="1.0"?>
2  <launch>
3
4    <param name="robot_description" command="cat '$(find m2wr_description)/urdf/m2wr.xacro'"/>
5
6    <!-- send fake joint values -->
7    <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
8      <param name="use_gui" value="False"/>
9    </node>
10
11   <!-- Combine joint values -->
```

```
12  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher"/>
13
14  <!-- Show in Rviz   -->
15  <node name="rviz" pkg="rviz" type="rviz" />
16
17 </launch>
```

To launch the project use the following command

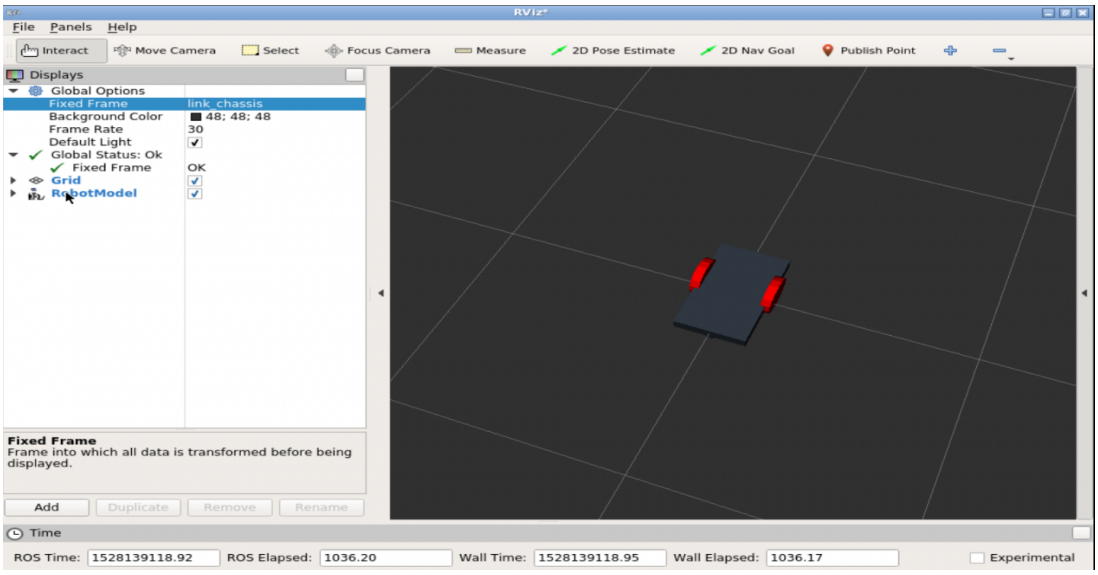
```
$ roslaunch m2wr_description rviz.launch
```

Once the node is launched we need to open **Graphical Tools** from the **Tools** menu, it will help us to see the rviz window.



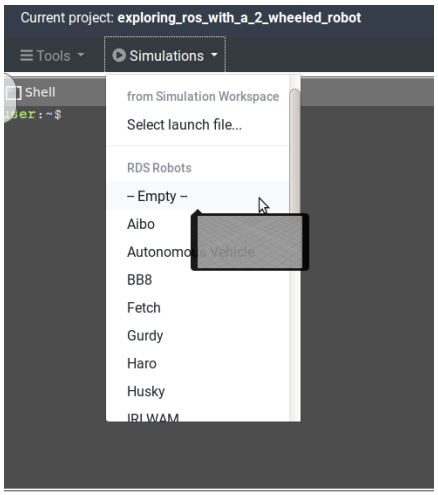
Once the rviz window loads, do the following:

- Change the frame to link_chasis in **Fixed Frame** option
- Add a Robot Description display



Step 1.4

We are now ready to simulate the robot in gazebo. We will load a **empty world** from the **Simulations** menu option



To load our robot into the **empty world** we will need another launch file. We will create another launch file with name **spawn.launch** inside the launch directory with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
    <param name="robot_description" command="cat '$(find m2wr_description)/urdf/m2wr.xacro'" />

    <arg name="x" default="0"/>
    <arg name="y" default="0"/>
    <arg name="z" default="0.5"/>

    <node name="mybot_spawn" pkg="gazebo_ros" type="spawn_model" output="screen"
          args="-urdf -param robot_description -model m2wr -x $(arg x) -y $(arg y) -z $(arg z)" />

</launch>
```

Next we will spawn our robot with the launch file in the empty gazebo world. Use the following command

```
$ roslaunch m2wr_description spawn.launch
```

The robot should load in the gazebo window

At this point our directory structure should look like following

```
.
├── ai_ws
├── catkin_ws
│   ├── build
│   ├── devel
│   └
├── notebook_ws
│   ├── default.ipynb
│   ├── images
└── simulation  ws
    ├── build
    ├── devel
    └── src
```

```
14      ├── CMakeLists.txt
15      └── m2wr_description
16          ├── CMakeLists.txt
17          ├── launch
18          │   ├── rviz.launch
19          │   └── spawn.launch
20          ├── package.xml
21          └── urdf
22              └── m2wr.xacro
```

Step 1.5

Now we are ready to add control to our robot. We will add a new element called **plugin** to our xacro file. We will add a differential drive **plugin** to our robot. The new tag looks like follows:

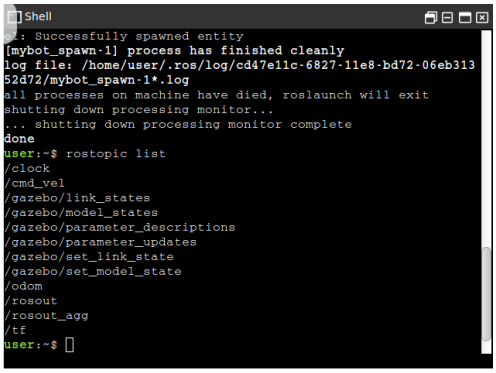
```
1  <gazebo>
2    <plugin filename="libgazebo_ros_diff_drive.so" name="differential_drive_controller">
3      <alwaysOn>true</alwaysOn>
4      <updateRate>20</updateRate>
5      <leftJoint>joint_left_wheel</leftJoint>
6      <rightJoint>joint  right  wheel</rightJoint>
7      <wheelSeparation>0.4</wheelSeparation>
8      <wheelDiameter>0.2</wheelDiameter>
9      <torque>0.1</torque>
10     <commandTopic>cmd_vel</commandTopic>
11     <odometryTopic>odom</odometryTopic>
12     <odometryFrame>odom</odometryFrame>
13     <robotBaseFrame>link_chassis</robotBaseFrame>
14   </plugin>
15 </gazebo>
```

Add this element inside the <robot> </robot> tag and re-launch the project. Now we will be able to control the robot, we can check this by listing the available topics using following command

```
$ rostopic list
```

To control the motion of the robot we can use the **keyboard_teleop** to publish motion commands using the keyboard. Use the following command in **Shell**

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



To control the motion of the robot we can use the **keyboard_teleop** to publish motion commands using the keyboard. Use the following command in **Shell**

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

Now we can make the robot navigate with **keyboard** keys. This finishes the part-1.

REFERENCES

RDS: HTTPS://RDS.THECONSTRUCTSIM.COM/

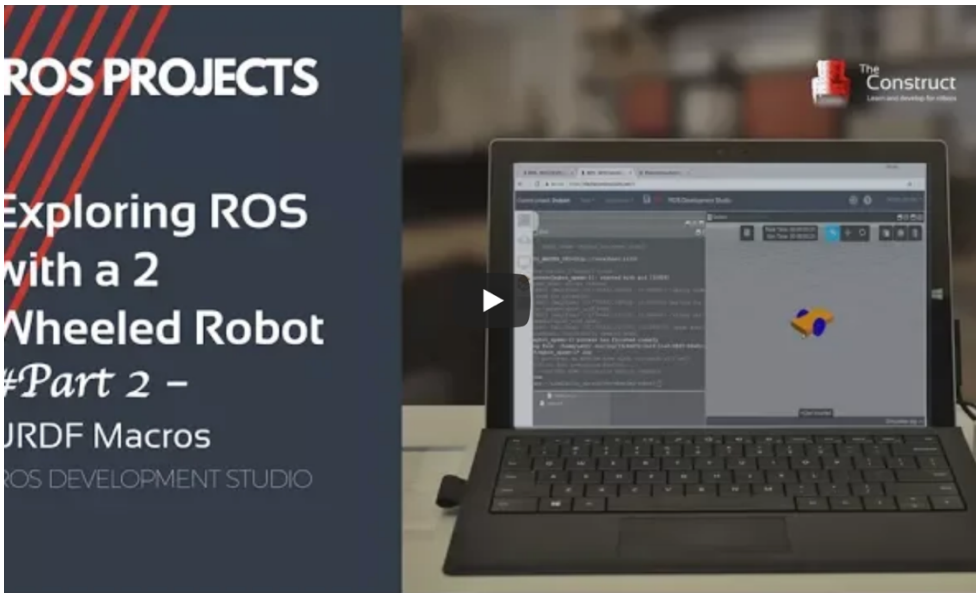
SOURCE CODE REPOSITORY: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT/

INERTIA MATRIX REFERENCE:
HTTPS://EN.WIKIPEDIA.ORG/WIKI/LIST_OF_MOMENTS_OF_INERTIA

Exploring ROS with a 2 Wheeled Robot #Part 2 - URDF Macros

In this video, we are going to explore the macros for URDF files, using XACRO files. At the end of this video, we will have the same model organized in different files, in a organized way.

Video Tutorial Part2



Step 2.1

In the last 5 steps we achieved the following:

- Created a xacro file that contains the urdf description of our robot
- Created a launch files to spawn the robot in gazebo environment
- Controlled the simulated robot using **keyboard teleoperation**

In this part we will organize the existing project to make it more readable and modular. Even though our robot description had only few components, we had written a lengthy xacro file. Also in robot spawn launch file we have used the following line

```
1 <param name="robot_description" command="cat '$(find m2wr_description)/urdf/m2wr.xacro'" />
```

Here we are using the cat command to **read** the contents of **m2wr.xacro** file into **robot_description** parameter. However, to use the features of the xacro file we need to parse and execute the xacro file and to achieve that we will modify the above line to

```
1 <param name="robot_description" command="$(find xacro)/xacro.py '$(find m2wr_description)/urdf
```

The above command, uses the xacro.py file to execute the instructions of **m2wr.xacro** file. A similar edit is needed for the **rviz.launch** file as well. So change the following line in **rviz.launch** file

```
1 <param name="robot_description" command="cat '$(find m2wr_description)/urdf/m2wr.xacro'" />
```

to

```
1 <param name="robot_description" command="$(find xacro)/xacro.py  '$(find m2wr_description)/urdf/m2wr.xacro'"/>
```

Step 2.2

At the moment, our xacro file does not contain any instructions. We will now split up the large **m2wr.xacro** file into smaller files and using the features of xacro we will assimilate the smaller files.

First we will extract the **material** properties from our xacro file and place them in a new file called **materials.xacro**. We will create a new file named **materials.xacro** inside the urdf folder and write the following contents into it

```
1  <?xml version="1.0" ?>
2  <robot name="m2wr" xmlns:xacro="https://www.ros.org/wiki/xacro" >
3    <material name="black">
4      <color rgba="0.0 0.0 0.0 1.0"/>
5    </material>
6    <material name="blue">
7      <color rgba="0.203125 0.23828125 0.28515625 1.0"/>
8    </material>
9    <material name="green">
10     <color rgba="0.0 0.8 0.0 1.0"/>
11   </material>
12   <material name="grey">
13     <color rgba="0.2 0.2 0.2 1.0"/>
14   </material>
15   <material name="orange">
16     <color rgba="1.0 0.423529411765 0.0392156862745 1.0"/>
```

```
17   </material>
18   <material name="brown">
19     <color rgba="0.870588235294 0.811764705882 0.764705882353 1.0"/>
20   </material>
21   <material name="red">
22     <color rgba="0.80078125 0.12890625 0.1328125 1.0"/>
23   </material>
24   <material name="white">
25     <color rgba="1.0 1.0 1.0 1.0"/>
26   </material>
27 </robot>
```

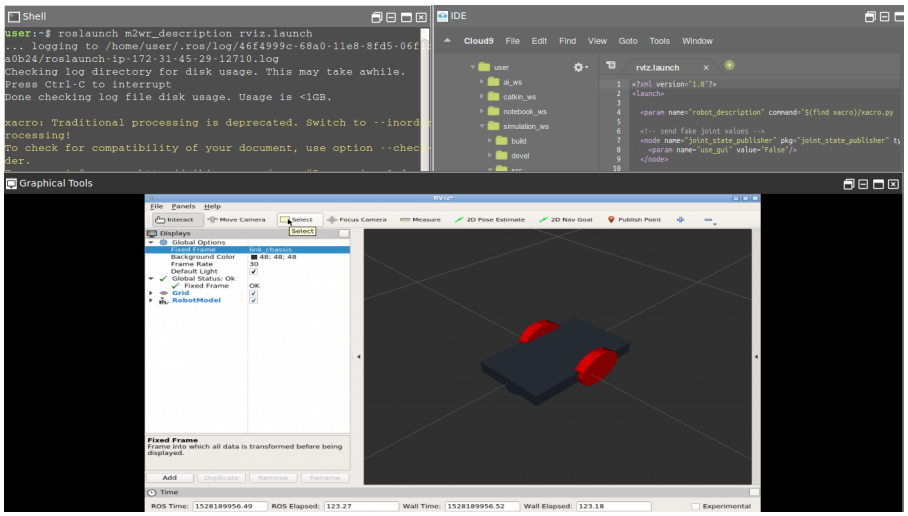
We need to replace all `material` elements in the original **m2wr.xacro** file with following `include` directive

```
1 <xacro:include filename="$(find m2wr_description)/urdf/materials.xacro" />
```

To test the changes, we can start the rviz visualization with command

```
$ roslaunch m2wr_description rviz.launch
```

Use **Graphical Tool** to see the rviz output



Going further we will now remove more code from the **m2wr.xacro** file and place it in a new file. Create a new file with name **m2wr.gazebo** inside the urdf directory. We will move all the gazebo tags from the **m2wr.xacro** file to this new file. We will need to add the enclosing

```
1 <?xml version="1.0" ?>
2 <robot name="m2wr" xmlns:xacro="https://www.ros.org/wiki/xacro" >
3 <gazebo reference="link_chassis">
```

```
    <material>Gazebo/Orange</material>
  </gazebo>
  <gazebo reference="link_left_wheel">
    <material>Gazebo/Blue</material>
  </gazebo>
  <gazebo reference="link_right_wheel">
    <material>Gazebo/Blue</material>
  </gazebo>

  <gazebo>
    <plugin filename="libgazebo_ros_diff_drive.so" name="differential_drive_controller">
      <alwaysOn>true</alwaysOn>
      <updateRate>20</updateRate>
      <leftJoint>joint_left_wheel</leftJoint>
      <rightJoint>joint_right_wheel</rightJoint>
      <wheelSeparation>0.4</wheelSeparation>
      <wheelDiameter>0.2</wheelDiameter>
      <torque>0.1</torque>
      <commandTopic>cmd_vel</commandTopic>
      <odometryTopic>odom</odometryTopic>
      <odometryFrame>odom</odometryFrame>
      <robotBaseFrame>link_chassis</robotBaseFrame>
    </plugin>
  </gazebo>
```

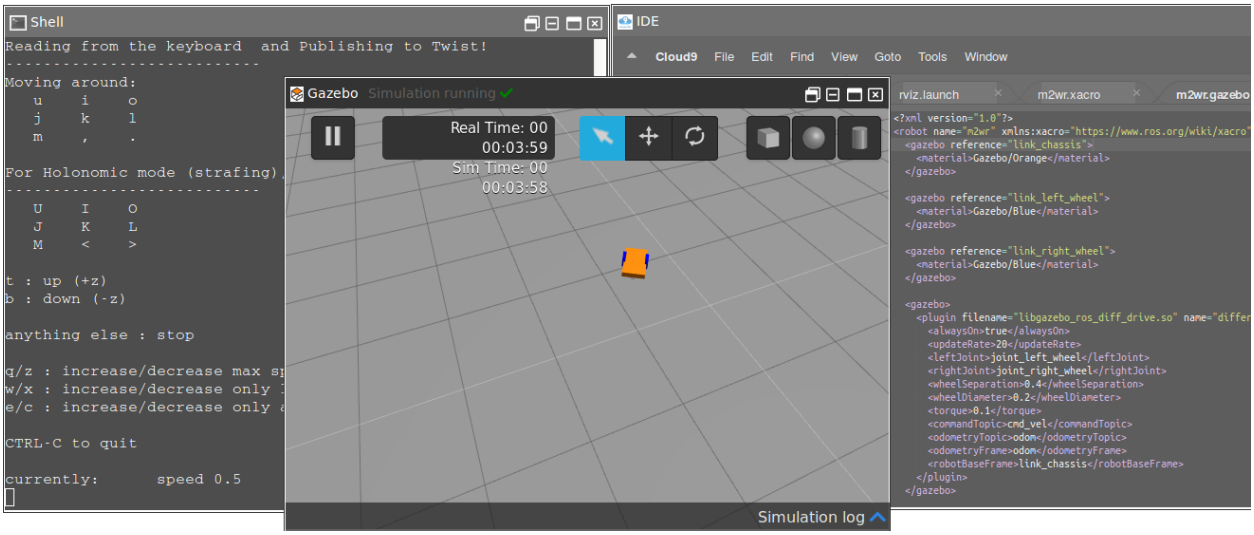
We will add another include directive to the **m2wr.xacro** file (as shown)

```
<xacro:include filename="$(find m2wr_description)/urdf/gazebo.xacro" />
```

To see whether everything works, we can launch the gazebo simulation of and **empty world** and **spawn** the robot. First we will start the gazebo simulation from the **Simulations** menu option and then spawn a robot with the following command

```
$ roslaunch m2wr_description spawn.launch
```

We should see the robot being spawned



Step 2.3

Next we will use macros, which are like functions, to reduce the remaining code in **m2wr.xacro** file.

Create a new file **macro.xacro** inside the urdf directory with following contents

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <xacro:macro name="link_wheel" params="name">
    <link name="${name}">
      <inertial>
        <mass value="0.2"/>
        <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
        <inertia ixx="0.000526666666667" ixy="0" ixz="0" iyy="0.000526666666667" iyz="0" izz="0.001"/>
      </inertial>
      <collision name="link_right_wheel_collision">
        <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
        <geometry>
          <cylinder length="0.04" radius="0.1"/>
        </geometry>
      </collision>
      <visual name="${name}_visual">
        <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
        <geometry>
          <cylinder length="0.04" radius="0.1"/>
        </geometry>
      </visual>
    </link>
  </xacro:macro>

  <xacro:macro name="joint_wheel" params="name child origin_xyz">
    <joint name="${name}" type="continuous">
      <origin rpy="0 0 0" xyz="${origin_xyz}"/>
```

```
28       <child link="${child}"/>
29       <parent link="link_chassis"/>
30       <axis rpy="0 0 0" xyz="0 1 0"/>
31       <limit effort="10000" velocity="1000"/>
32       <joint_properties damping="1.0" friction="1.0"/>
33     </joint>
34   </xacro:macro>
35
36 </robot>
```

In the above code we have defined three macros, their purpose is to take parameters and create the required element (link element). The first macro is named **link_wheel** and it accepts only one parameter **name**. It creates a wheel link with the name passed in the parameter. The second macro accepts three parameters **name**, **child** and **origin_xyz** and it creates a **joint** link.

We will use these macro in our robot description file (**m2wr.xacro**). To use macros we will replace the link element by the macros as follows

```
1   <link name="link_right_wheel">
2     <inertial>
3       <mass value="0.2"/>
4       <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
5       <inertia ixx="0.00052666666" ixy="0" ixz="0" iyy="0.00052666666" iyz="0" izz="0.001"/>
6     </inertial>
7
8     <collision name="link_right_wheel_collision">
9       <origin rpy="0 1.5707 1.5707" xyz="0 0 0" />
10      <geometry>
11        <cylinder length="0.04" radius="0.1"/>
12      </geometry>
13    </collision>
14
15    <visual name="link_right_wheel_visual">
16      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
17      <geometry>
18        <cylinder length="0.04" radius="0.1"/>
19      </geometry>
20    </visual>
21  </link>
22
23  <link name="link_left_wheel">
24    <inertial>
25      <mass value="0.2"/>
26      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
27      <inertia ixx="0.00052666666" ixy="0" ixz="0" iyy="0.00052666666" iyz="0" izz="0.001"/>
28    </inertial>
29
30    <collision name="link_left_wheel_collision">
31      <origin rpy="0 1.5707 1.5707" xyz="0 0 0" />
32      <geometry>
```

```
33        <cylinder length="0.04" radius="0.1"/>
34      </geometry>
35    </collision>
36
37    <visual name="link_left_wheel_visual">
38      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
39      <geometry>
40        <cylinder length="0.04" radius="0.1"/>
41      </geometry>
42    </visual>
43  </link>
```

replaced by

```
1 <xacro:link_wheel name="link_right_wheel" />
2 <xacro:link_wheel name="link_left_wheel" />
```

Similar addition for the **link_left_wheel**. We will also replace the two wheel joint elements with following

```
1 <xacro:joint_wheel name="joint_right_wheel"  child="link_right_wheel"  origin_xyz="-0.05 0.15 0"  />
2 <xacro:joint_wheel name="joint_left_wheel"   child="link_left_wheel"   origin_xyz="-0.05 -0.15 0"  />
```

We also need to add the `include` directive for the **macro.xacro** file (in **m2wr.xacro** file)

```
1 <xacro:include filename="$(find m2wr_description)/urdf/macro.xacro" />
```

The final contents of the **m2wr.xacro** should be following

```
1   <?xml version="1.0" ?>
2   <robot name="m2wr" xmlns:xacro="https://www.ros.org/wiki/xacro" >
3
4    <!-- include the xacro files-->
5    <xacro:include filename="$(find m2wr_description)/urdf/materials.xacro" />
6    <xacro:include filename="$(find m2wr_description)/urdf/m2wr.gazebo" />
7    <xacro:include filename="$(find m2wr_description)/urdf/macro.xacro" />
8
9    <!-- Chasis defined here -->
10   <link name="link_chassis">
11    <pose>0 0 0.1 0 0 0</pose>
12    <inertial>
13      <mass value="5"/>
14      <origin rpy="0 0 0" xyz="0 0 0.1"/>
15      <inertia ixx="0.0395416666667" ixy="0" ixz="0" iyy="0.106208333333" iyz="0" izz="0.106208333333"/>
16    </inertial>
17
18    <collision name="collision_chassis">
19      <geometry>
```

```
      <box size="0.5 0.3 0.07"/>
    </geometry>
  </collision>

  <visual>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <geometry>
      <box size="0.5 0.3 0.07"/>
    </geometry>
    <material name="blue"/>
  </visual>

  <!-- caster front -->
  <collision name="caster_front_collision">
    <origin rpy=" 0 0 0" xyz="0.35 0 -0.05"/>
    <geometry>
      <sphere radius="0.05"/>
    </geometry>
    <surface>
      <friction>
        <ode>
          <mu>0</mu>
          <mu2>0</mu2>
          <slip1>1.0</slip1>
          <slip2>1.0</slip2>
        </ode>
      </friction>
    </surface>
  </collision>
  <visual name="caster_front_visual">
    <origin rpy=" 0 0 0" xyz="0.2 0 -0.05"/>
    <geometry>
      <sphere radius="0.05"/>
    </geometry>
  </visual>
  </link>


 <!-- Create wheel right -->

  <xacro:link_wheel name="link_right_wheel" />
  <xacro:joint_wheel name="joint_right_wheel"  child="link_right_wheel"  origin_xyz="-0.05 0.15 0"  />


 <!-- Left Wheel link -->

 <xacro:link_wheel name="link_left_wheel" />
 <xacro:joint_wheel name="joint_left_wheel"   child="link_left_wheel"   origin_xyz="-0.05 -0.15 0"  />

</robot>
```

The above code is much shorter than what we started with. We have only used a few of the features available in xacro description.

Finally, we can test everything together by launching the gazebo simulator and spawning our robot. Start a new gazebo simulator with **empty world** and spawn our robot

```
$ roslaunch m2wr_description spawn.launch
```

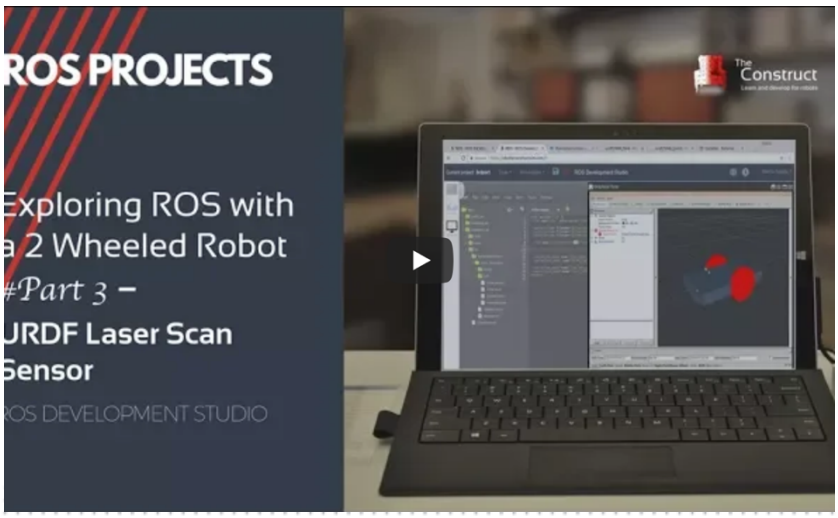
That is it, we have optimized our code by splitting the large xacro file into other files and using macros.

REFERENCES
RDS: HTTPS://RDS.THECONSTRUCTSIM.COM/
SOURCE CODE REPOSITORY: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT

Exploring ROS with a 2 Wheeled Robot #Part 3 - URDF Laser Scan Sensor

In this video, we are going to insert a laser scan sensor to a 2 wheeled robot the robot.

Video Tutorial Part3



Step 3.1

Now we will add a **laser scan sensor** to our robots urdf model. We will modify the urdf file (**m2wr.xacro**) as follows

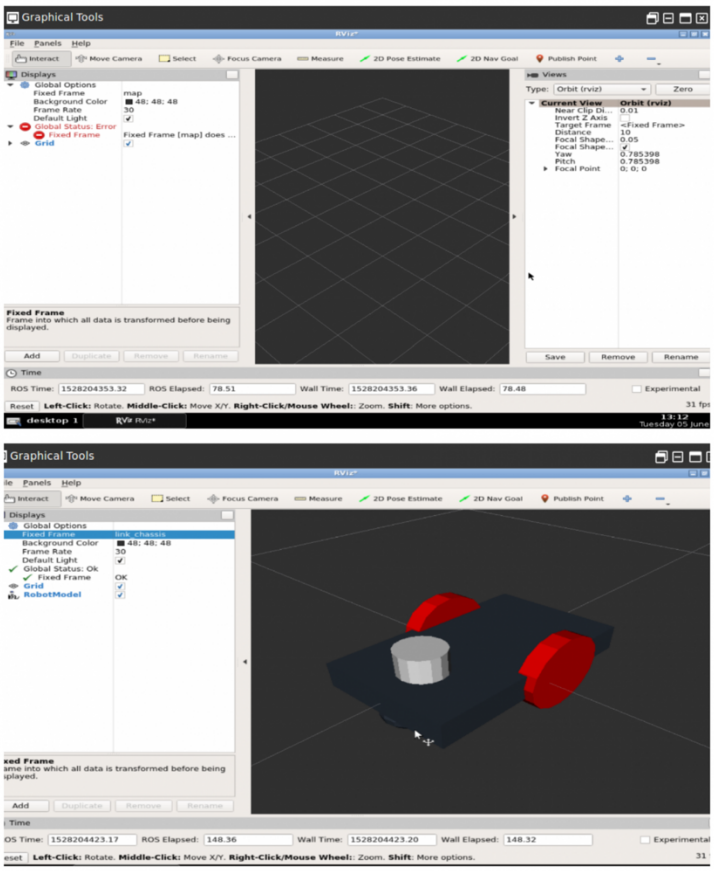
- Add a link element to our robot. This link will be ***cylindrical*** in shape and will represent the sensor.

- Add a joint element to our robot. This will connect the sensor to robot body rigidly.
- Define a new macro to calculate the inertial property of a cylinder using its dimensions (length and radius)
- Finally, a **laser scan sensor** plugin element will add sensing ability to the link that we created (the cylinder representing the sensor).

Open the **m2wr.xacro** file and add a new link element and a new joint element

```
<link name="sensor_laser">
  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <mass value="1" />
    <!-- RANDOM INERTIA BELOW -->
    <inertia ixx="0.02" ixy="0" ixz="0" iyy="0.02" iyz="0" izz="0.02"/>
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <cylinder radius="0.05" length="0.1"/>
    </geometry>
    <material name="white" />
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="0.05" length="0.1"/>
    </geometry>
  </collision>
</link>

<joint name="joint_sensor_laser" type="fixed">
  <origin xyz="0.15 0 0.05" rpy="0 0 0"/>
  <parent link="link_chassis"/>
  <child link="sensor_laser"/>
</joint>
```

The above code block will result in the following visualization



Step 3.2

The link element we just added (for acting as sensor) has random inertia property (see line 5 of the above code). We can write some sane values using a macro that will calculate the inertia values using cylinder dimensions. For this we add a new macro to our macro.xacro script. Add the following macro to the file

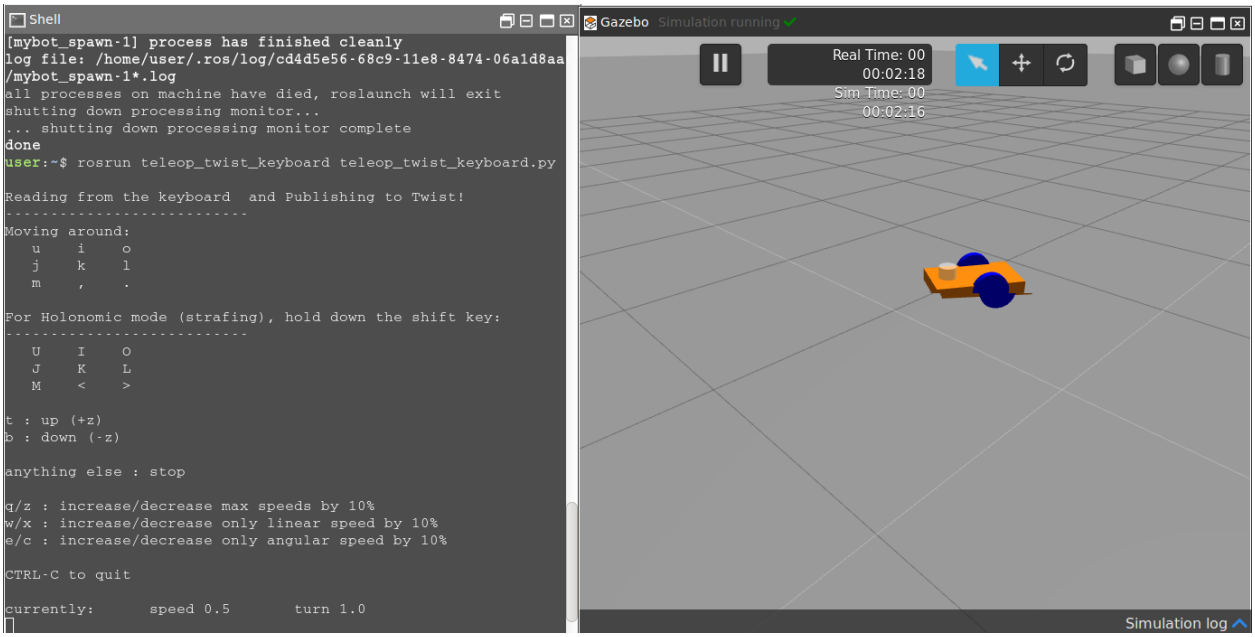
```
1 <xacro:macro name="cylinder_inertia" params="mass r l">
2    <inertia ixx="${mass*(3*r*r+l*l)/12}" ixy = "0" ixz = "0"
3             iyy="${mass*(3*r*r+l*l)/12}" iyz = "0" izz="${mass*(r*r)/2}" />
4 </xacro:macro>
```

Let's use this `macro` in the sensor description `link` by modifying the **inertia** tag with following

```
1  <xacro:cylinder_inertia mass="1" r="0.05" l="0.1" />
```

Now we can simulate the robot and see if everything works well. Load an **empty world** in **gazebo simulator** window. Also open a **Shell** window and run the following command

```
$ roslaunch m2wr_description spawn.launch
```



Next we need to add the sensor behavior to the link. To do so we will use the laser gazebo plugin. Information about this plugin is available here. Open the **m2wr.gazebo** file and add the following **plugin element**

```
1  <gazebo reference="sensor_laser">
2    <sensor type="ray" name="head_hokuyo_sensor">
3      <pose>0 0 0 0 0 0</pose>
4      <visualize>false</visualize>
5      <update_rate>20</update_rate>
6      <ray>
7        <scan>
8          <horizontal>
9            <samples>720</samples>
10           <resolution>1</resolution>
11           <min_angle>-1.570796</min_angle>
12           <max_angle>1.570796</max_angle>
13         </horizontal>
14       </scan>
15       <range>
```

```
      <min>0.10</min>
      <max>10.0</max>
      <resolution>0.01</resolution>
    </range>
    <noise>
      <type>gaussian</type>
      <mean>0.0</mean>
      <stddev>0.01</stddev>
    </noise>
  </ray>
  <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
    <topicName>/m2wr/laser/scan</topicName>
    <frameName>sensor_laser</frameName>
  </plugin>
 </sensor>
</gazebo>
```

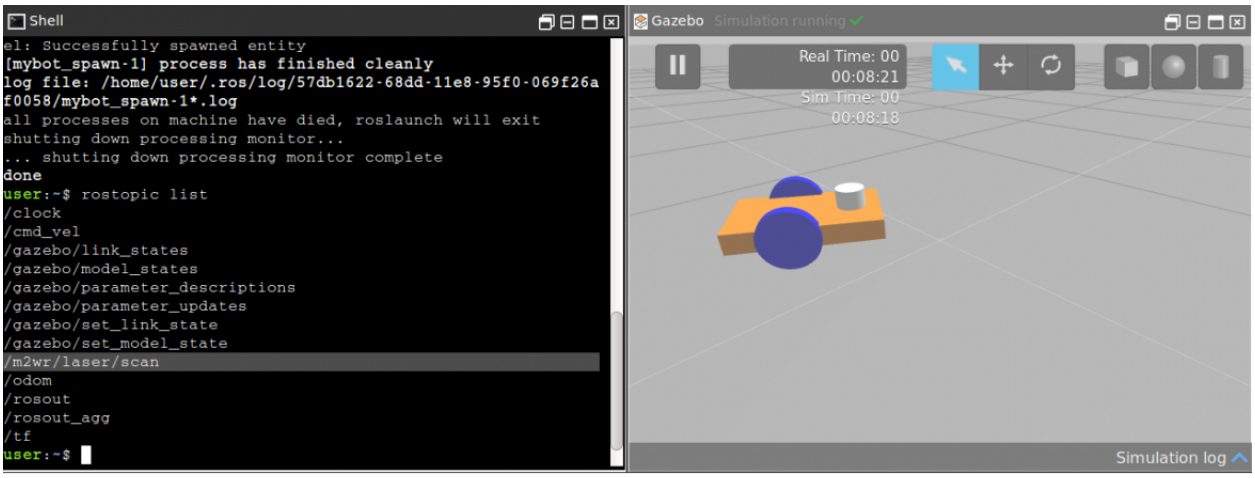
This code specifies many important parameters

- **Update rate**: Controls how often (how fast) the laser data is captured
- **samples**: Defines how how many readings are contained in one scan
- **resolution**: Defines the minimum angular distance between readings captured in a laser scan
- **range**: Defines the minimum sense distance and maximum sense distance. If a point is below minimum sense its reading becomes zero (`0`) and if a point is further than the maximum sense distance its reading becomes `inf`. The **range resolution** defines the minimum distance between 2 points such that two points can be resolved as two separate points.
- **noise**: This parameter lets us add gaussian noise to the range data captured by the sensor
- **topicName**: Defines the name which is used for publishing the laser data
- **frameName**: Defines the link to which the plugin has to be applied

With this plugin incorporated in the urdf file we are now ready to simulate and visualize the laser scan in action. Start the **empty world** and spawn the robot by launching the **spawn.launch** file in a **Shell**. To verify the working of the scan sensor check the list of topics in **Shell** window with following command

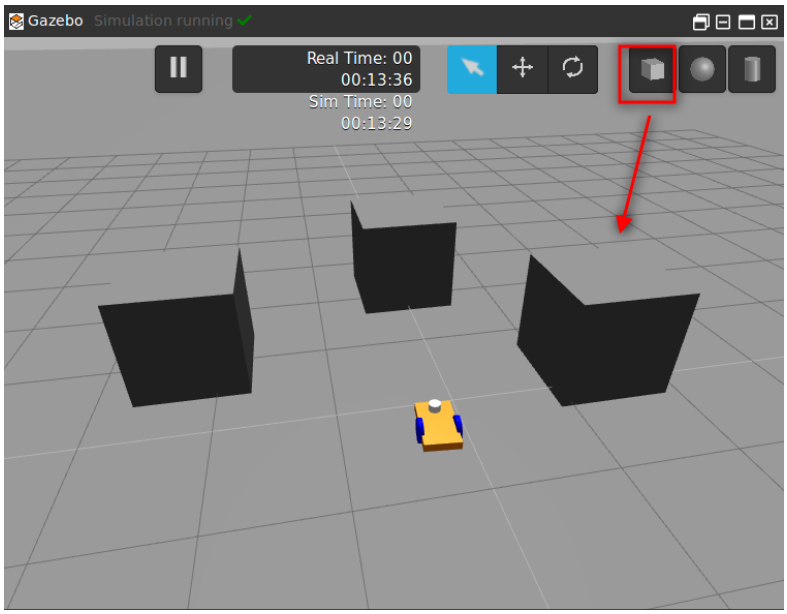
```
$ rostopic list
```

You should get **/m2wr/laser/scan** in the list of topics



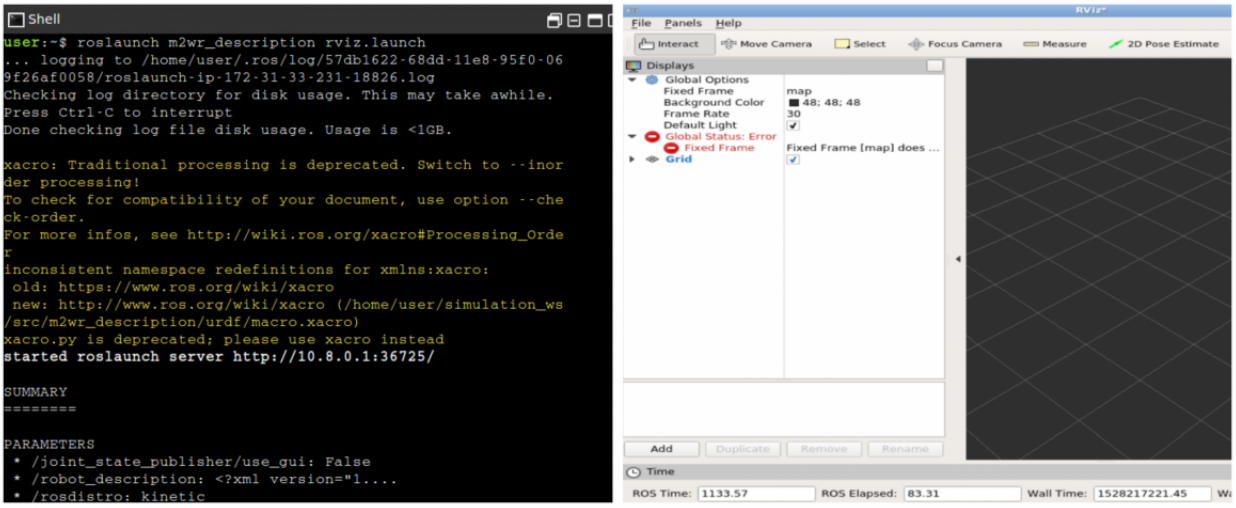
Step 3.3

We will visualize the laser scan data with rviz. First we will populate the robot environment with a few obstacles to better see the laser scan result. Use the box icon on top right of **gazebo** window to create a few box type obstacles (simply click and drop)



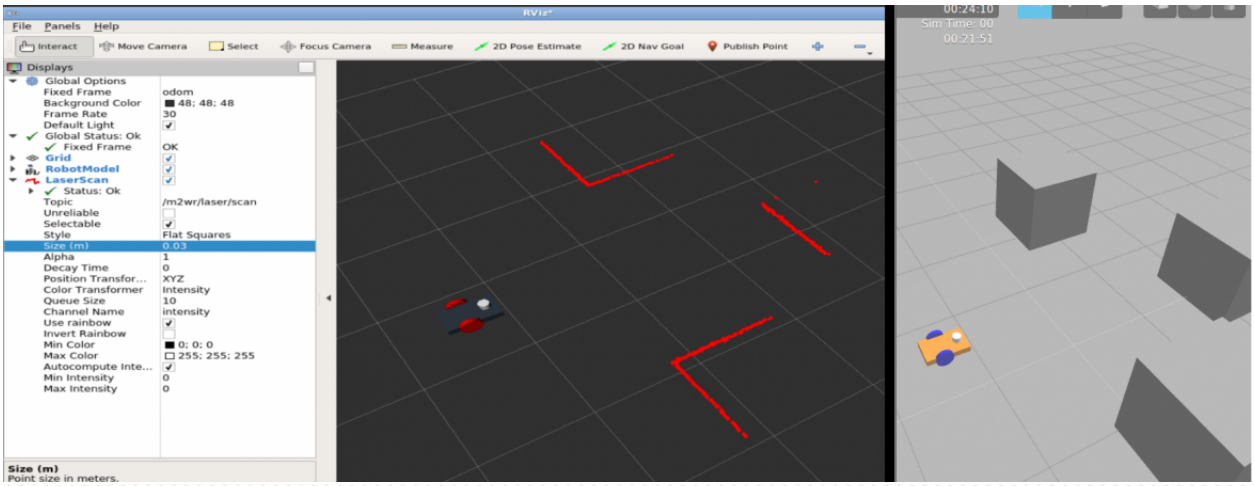
To start rviz visualization launch the **rviz.launch** file in a new **Shell** and use **Graphical Tool** window to load the visualization. Use the following command to launch rviz

```
$ roslaunch m2wr_description rviz.launch
```



After starting rviz open the **Graphical Tools** window. Once rviz window loads you need to do the following settings

- Select **odom** in the **Fixed Frame** field (see the image below)
- Add two new displays using the **Add** button on the left bottom of rviz screen. The first display should be RobotModel and the other should be **LaserScan**
- Expand the **LaserScan** display by double clicking on its name and choose **Topic** as **/m2wr/laser/scan** (as shown in image below)



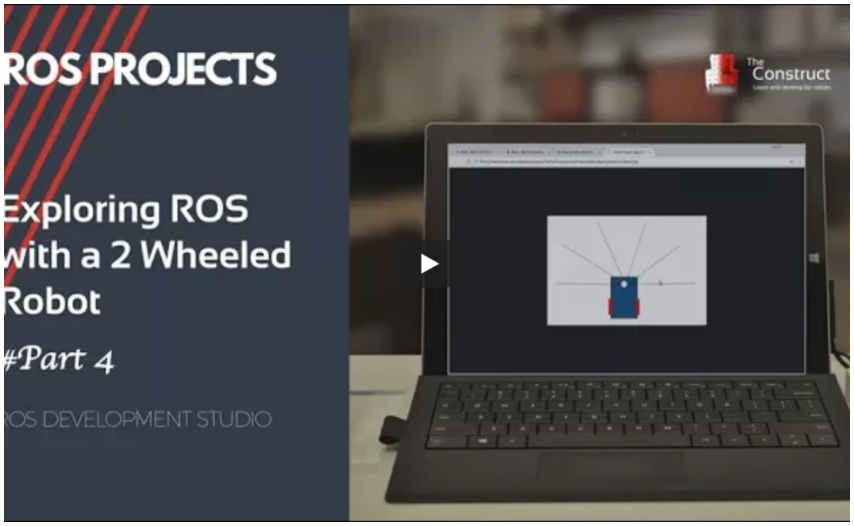
Now when we move the robot we can see the laser scan changing.

References
RDS: https://rds.theconstructsim.com/
Source Code Repository: https://bitbucket.org/theconstructcore/two-wheeled-robot
Gazebo plugins: http://gazebosim.org/tutorials?tut=ros_gzplugins#Laser
ROS URDF Links: http://wiki.ros.org/urdf/XML/link
ROS URDF Joints: http://wiki.ros.org/urdf/XML/joint

Exploring ROS using a 2 Wheeled Robot - Part 4

In this video, we are going to read the values of the laser scanner and filter a small part to work with

Video Tutorial Part 4



Step 4.1

- Head to ROS Development Studio and create a new project. Provide a suitable project name and some useful description. (We have named the project **video_no_4**)

- Open the project (this will take few seconds).

- We will clone the *github* repository to start. Open a **Shell** from the **Tools** menu and run the following commands in the **Shell**

```
$ cd simulation_ws/src

$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-
wheeled-robot-simulation.git
```

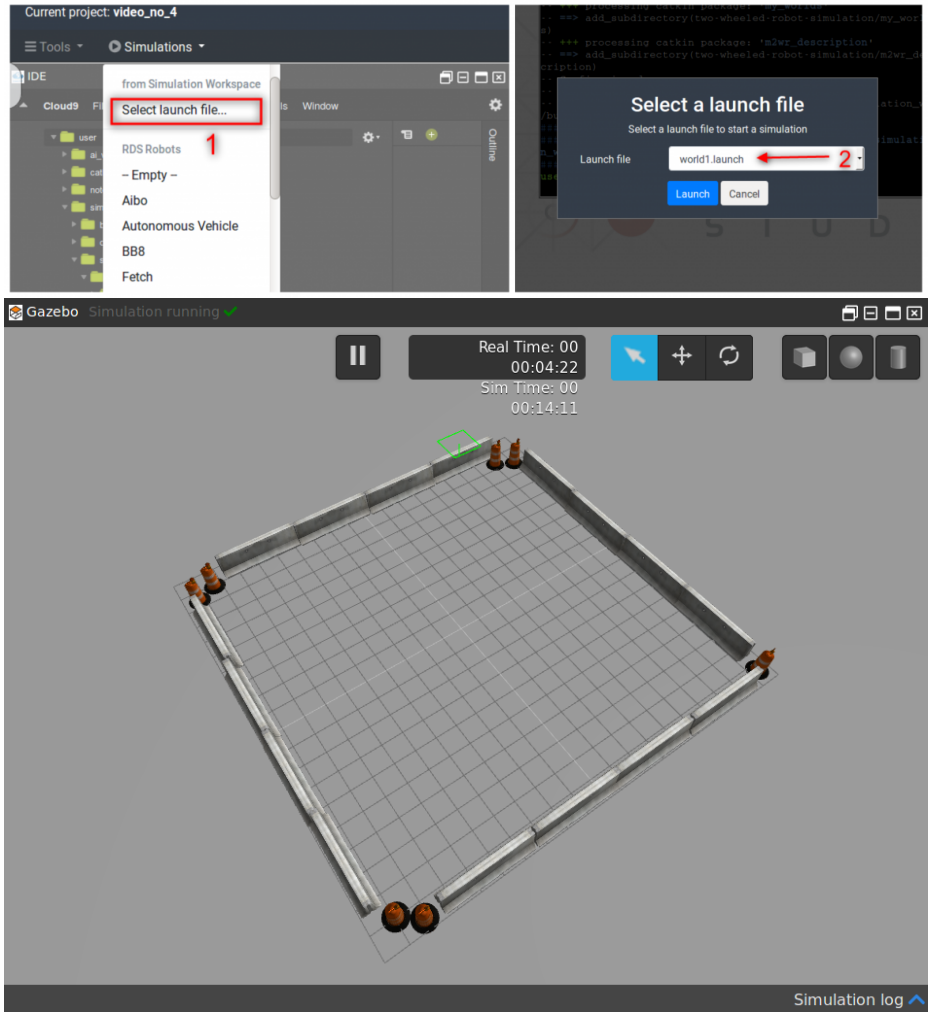
- We should have the following directory structure at this point

```
.
├──────── ai_ws
├──────── catkin_ws
|     ├── build
|     ├── devel
|     └──
├──────── notebook_ws
|     ├── default.ipynb
|     └── images
└──────── simulation_ws
      ├── build
      ├── devel
      └── src
          ├── m2wr_description
          |   ├── CMakeLists.txt
          |   ├── launch
          |   ├── package.xml
          |   └── urdf
          └── my_worlds
              ├── CMakeLists.txt
              ├── launch
              ├── package.xml
              └── worlds
```

The package **m2wr_description** contains the project files developed so far (i.e. robot + laser scan sensor). The other package **my_worlds** contains two directories

- **launch**: Contains a launch file (we will use it shortly)
- **worlds**: Contains multiple world description files

From **Simulations** menu, choose the option **Select launch file...** and select the **world1.launch** option



Step 4.2

We will create a new catkin package named **motion_plan** with dependencies **rospy**, **std_msgs**, **geometry_msgs** and **sensor_msgs**. Open a **Shell** from the **Tools** menu and write the following commands

```
$ cd ~/catkin_ws/src $ catkin_create_pgk motion_plan rospy std_msgs geometry_msgs sensor_msgs $ cd motion_plan $ mkdir scripts $ touch scripts/reading_laser.py
```

These commands will create a directory (named **scripts**) inside the **motion_plan** package. This directory will contain a python scripts (**reading_laser.py**) that we will use to read the laser scan data coming on the **/m2wr/laser/scan** topic (we created this topic in last part). Add the following code to the **reading_laser.py** file

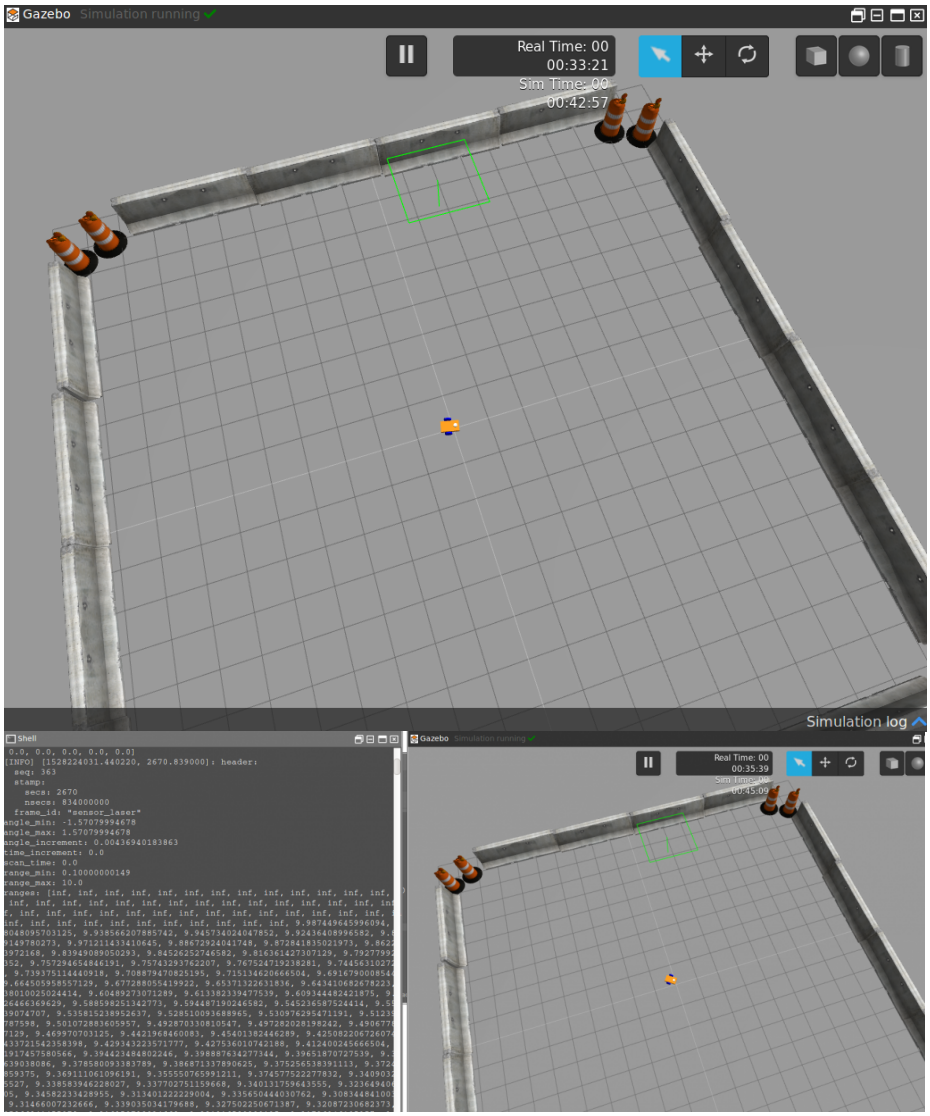
```
#! /usr/bin/env python

import rospy
from sensor_msgs.msg import LaserScan

def clbk_laser(msg):
    # 720/5 = 144
    regions = [
      min(msg.ranges[0:143]),
      min(msg.ranges[144:287]),
      min(msg.ranges[288:431]),
      min(msg.ranges[432:575]),
      min(msg.ranges[576:713]),
    ]
    rospy.loginfo(regions)

def main():
    rospy.init_node('reading_laser')
    sub= rospy.Subscriber("/m2wr/laser/scan", LaserScan, clbk_laser)

    rospy.spin()

if __name__ == '__main__':
    main()
```

We will make this script executable with following commands

```
$ cd ~/catkin_ws/src/motion_plan/scripts/ $ chmod +x reading_laser.py
```

Before we run this file, we need to spawn our robot into the gazebo simulation. Use the following commands

```
$ roslaunch m2wr_description spawn.launch $ rosrun motion_plan reading_laser.py
```

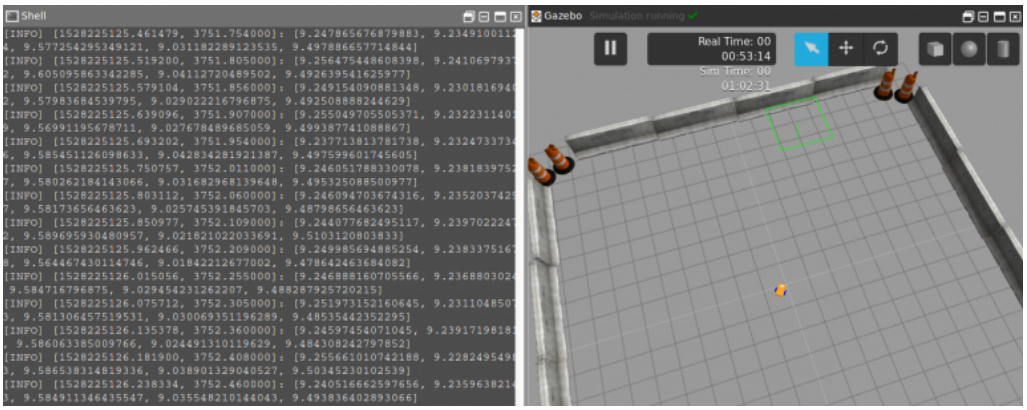


Once we have verified that everything is working fine. We will modify the callback function to following:

```
1  def clbk_laser(msg):
2      # 720/5 = 144
3      regions = [
4        min(msg.ranges[0:143]),
5        min(msg.ranges[144:287]),
6        min(msg.ranges[288:431]),
7        min(msg.ranges[432:575]),
8        min(msg.ranges[576:713]),
9        ]
10     rospy.loginfo(regions)
```

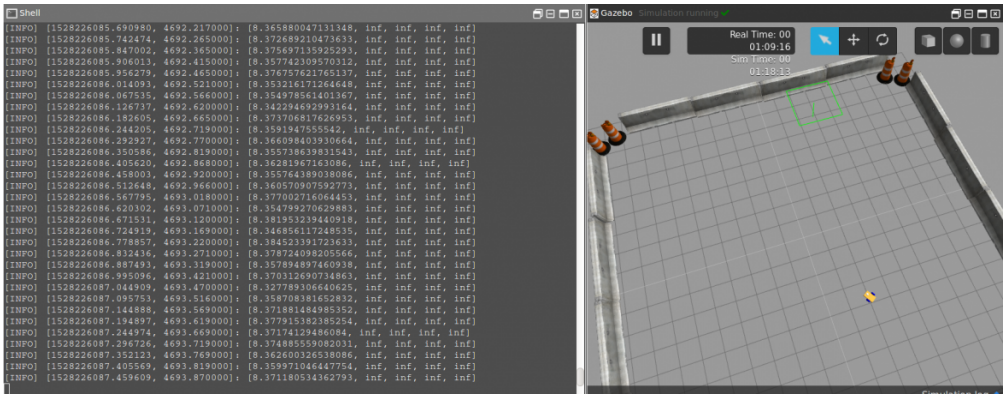
The above code converts the 720 readings contained inside the `LaserScan` msg into five distinct readings. Each reading is the minimum distance measured on a sector of 60 degrees (total 5 sectors = 180 degrees).

Let's run this code again and see the data



Step 4.3

We can see the changes in range measurements as we move the robot near to the boundaries. One noticeable thing is the inf value that is measured when the wall is away.

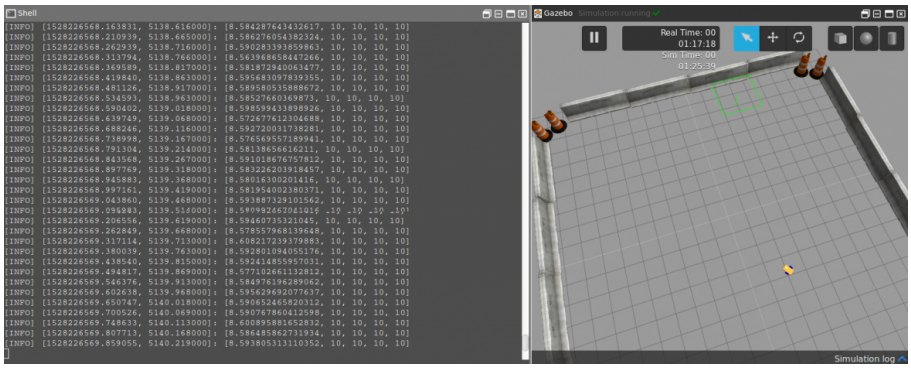


We can modify the code to change this value to read maximum range. The changed code is

```
1  def clbk_laser(msg):
2      # 720/5 = 144
```

```
3      regions = [
4        min(min(msg.ranges[0:143]), 10),
5        min(min(msg.ranges[144:287]), 10),
6        min(min(msg.ranges[288:431]), 10),
7        min(min(msg.ranges[432:575]), 10),
8       min( min(msg.ranges[576:713]), 10),
9        ]
10     rospy.loginfo(regions)
```

With this modification we will only receive a value of range between 0 and 10.



Then we finish the part 4.

REFERENCES

RDS: HTTPS://RDS.THECONSTRUCTSIM.COM/

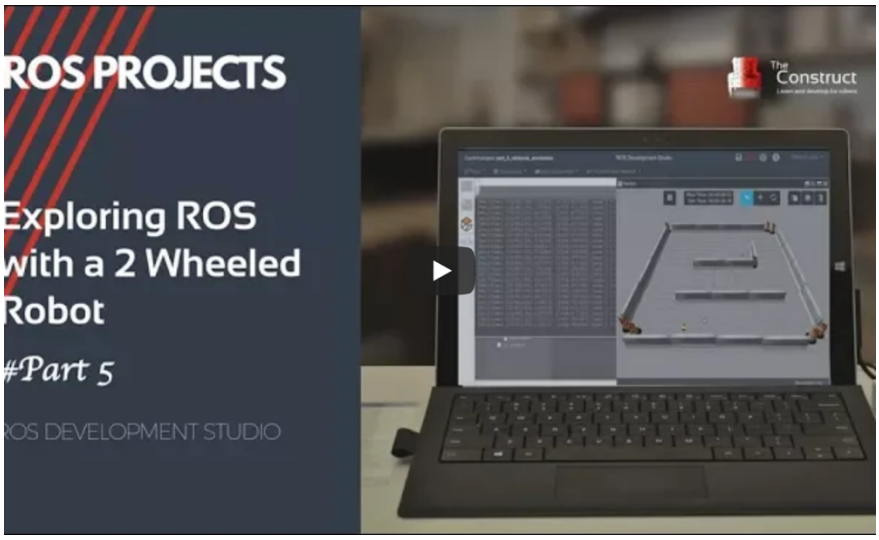
SOURCE CODE REPOSITORY: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT

GAZEBO PLUGINS: HTTP://GAZEBOSIM.ORG/TUTORIALS?TUT=ROS_GZPLUGINS#LASER

Exploring ROS with a 2 Wheeled Robot - Part 5

In this video an obstacle avoidance algorithm is explained an shown how it works.

Video Tutorial Part 5



Step 5.1

- Head to ROS Development Studio and create a new project.
- Provide a suitable project name and some useful description. (We have named the project **part 5-obstacle avoidance**)
- Load/Start the project (this will take few seconds).
- Clone the github repository two-wheeled-robot - Simulation.
- Checkout to the **right** branch (here is more information about branch and branching in git)
- Open a **Shell** from the **Tools** menu and run the following commands in the **Shell**

```
$ cd simulation_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-simulation.git
$ cd two-wheeled-robot-simulation
$ git checkout 16e45ce
```

- Compile the project to make it ready to use

```
cd ~/simulation_ws/src
catkin_make
```

- Clone another github repository [two-wheeled-robot - Motion Planning](). Execute the following commands in the **Shell**

```
$ cd catkin_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning.git
```

- Compile the project to make it ready to use

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

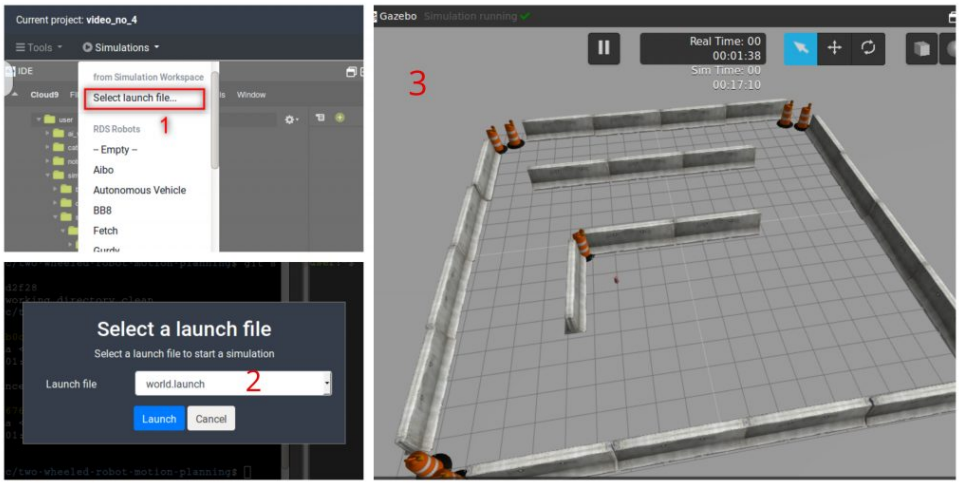
- At this point we should have the following directory structure

```
.
├──────── ai_ws
├──────── catkin_ws
|     ├─── build/
|     ├─── devel/
|     └─── src
|        ├── CMakeLists.txt
|        └── two-wheeled-robot-motion-planning
|           ├── CMakeLists.txt
|           ├── examples/
|           ├── launch/
|           ├── package.xml
|           └── scripts/
├──────── notebook_ws
|     ├─── default.ipynb
|     └─── images/
└──────── simulation_ws
    ├─── build/
    ├─── devel/
    └─── src
       ├── m2wr_description
       |   ├── CMakeLists.txt
       |   ├── launch/
       |   ├── package.xml
       |   └── urdf/
       └── my_worlds
```

```
28          ├── CMakeLists.txt
29          ├── launch/
30          |   └── world.launch
31          ├── package.xml
32          └── worlds/
33              ├── world01.world
                └── world02.world
```

Step 5.2

Start a simulation using the **Simulations** menu option. Select the **world.launch** option and launch the simulation.



Now we will take a look at the obstacle avoidance algorithm. Open the **IDE** and browse to the file **~/catkin_ws/src/two-wheeled-robot-motion-planning/scripts/obstacle_avoidance.py**. There are 3 functions defined in this file:

- main
  This is the entry point of the file. This function sets up a **Subscriber** to the laser scan topic **/m2wr/laser/scan** and a **Publisher** to **/cmd_vel** topic.

```
1 def main():
2 global pub
3
4 rospy.init_node('reading_laser')
5 pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
6 sub = rospy.Subscriber('/m2wr/laser/scan', LaserScan, clbk_laser)
```

```
7 rospy.spin()
```

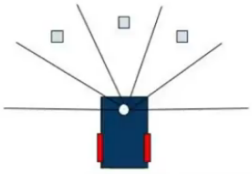
- clbk_laser
  We need to provide a callback function to the **Subscriber** defined in main, for this purpose we have this function. It receives laser scan data comprising of 720 readings and converts it into 5 readings (details in part 4 video).

```
1  def clbk_laser(msg):
2  regions = {
3  'right':  min(min(msg.ranges[0:143]), 10),
4  'fright': min(min(msg.ranges[144:287]), 10),
5  'front':  min(min(msg.ranges[288:431]), 10),
6  'fleft':  min(min(msg.ranges[432:575]), 10),
7  'left':   min(min(msg.ranges[576:713]), 10),
8  }
9
10 take_action(regions)
```

- take_action
  This function implements the obstacle avoidance logic. Based on the distances sensed in the five region (left, center-left, center, center-right, right). We consider possible combinations for obstacles, once we identify the obstacle configuration we steer the robot away from obstacle.

```
1  def take_action(regions):
2      msg = Twist()
3      linear_x = 0
4      angular_z = 0
5
6      state_description = ''
7
8      if regions['front'] > 1 and regions['fleft'] > 1 and regions['fright'] > 1:
9          state_description = 'case 1 - nothing'
10         linear_x = 0.6
11         angular_z = 0
12     elif regions['front'] < 1 and regions['fleft'] > 1 and regions['fright'] > 1:
13         state_description = 'case 2 - front'
14         linear_x = 0
15         angular_z = 0.3
16     elif regions['front'] > 1 and regions['fleft'] > 1 and regions['fright'] < 1:
17         state_description = 'case 3 - fright'
18         linear_x = 0
19         angular_z = 0.3
20     elif regions['front'] > 1 and regions['fleft'] < 1 and regions['fright'] > 1:
21         state_description = 'case 4 - fleft'
22         linear_x = 0
23         angular_z = -0.3
```

```
24    elif regions['front'] < 1 and regions['fleft'] > 1 and regions['fright'] < 1:
25        state_description = 'case 5 - front and fright'
26        linear_x = 0
27        angular_z = 0.3
28    elif regions['front'] < 1 and regions['fleft'] < 1 and regions['fright'] > 1:
29        state_description = 'case 6 - front and fleft'
30        linear_x = 0
31        angular_z = -0.3
32    elif regions['front'] < 1 and regions['fleft'] < 1 and regions['fright'] < 1:
33        state_description = 'case 7 - front and fleft and fright'
34        linear_x = 0
35        angular_z = 0.3
36    elif regions['front'] > 1 and regions['fleft'] < 1 and regions['fright'] < 1:
37        state_description = 'case 8 - fleft and fright'
38        linear_x = 0.3
39        angular_z = 0
40    else:
41        state_description = 'unknown case'
42        rospy.loginfo(regions)
43
44    rospy.loginfo(state_description)
45    msg.linear.x = -linear_x
46    msg.angular.z = angular_z
47    pub.publish(msg)
```



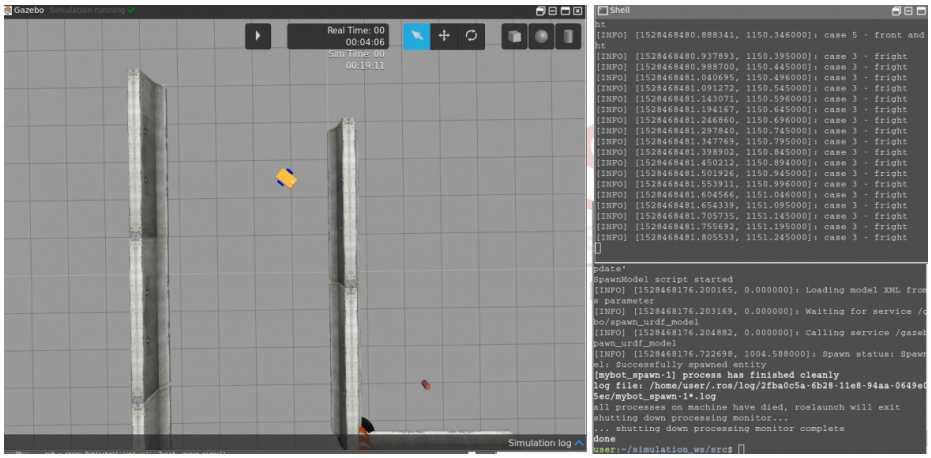
	Front Left	Front	Front Right
Case 1			
Case 2		x	
Case 3			x
Case 4	x		
Case 5		x	x
Case 6	x	x	
Case 7	x	x	x
Case 8	x		x

To test the logic lets run the simulation. We have the world loaded, now we will spawn the differential drive robot with following command

```
$ roslaunch m2wr_description spawn.launch
```

Finally, we launch the **obstacle avoidance** script to move the robot around and avoid obstacles

```
$ rosrun motion_plan obstacle_avoidance.py
```



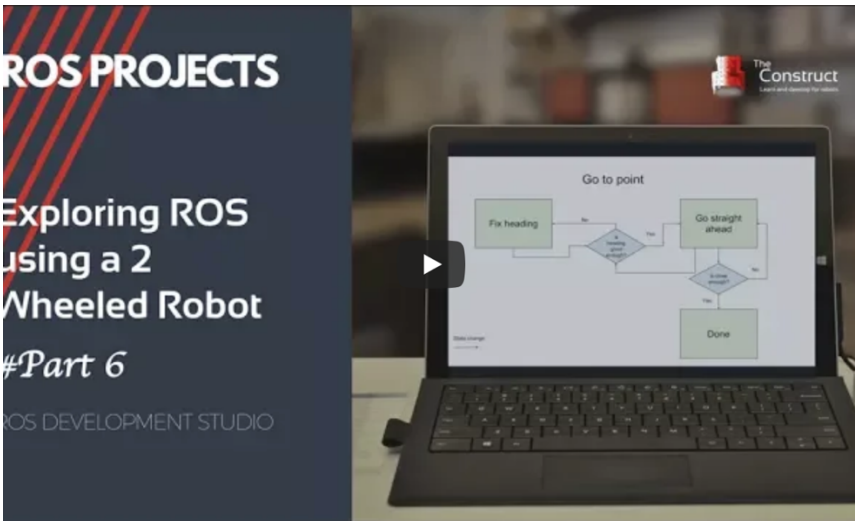
That finishes the instructions. You can change various settings like speed of robot, sensing distance, etc., and see how it works.

REFERENCES
RDS: HTTPS://RDS.THECONSTRUCTSIM.COM
PUBLIC REPOSITORIES:
SIMULATION: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT-SIMULATION
MOTION PLANNING: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT-MOTION-PLANNING

Exploring ROS with a 2 Wheeled Robot - Part 6

In this video, we are going create an algorithm to go from a point to another using the odometry data to localize the robot.

Video Tutorial Part 6



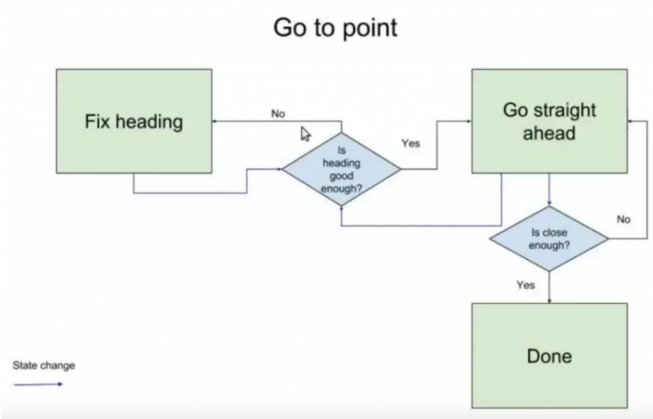
Steps to recreate the project as shown in the video (continued from part 5)

Step 6.1

In this part we are going to implement a simple navigation algorithm to move our robot from **any point** to a desired point. We will use the concept of state machines to implement the navigation logic. In a **state machine** there are finite number of states that represent the current situation (or behavior) of the system. In our case, we have **three** states

- **Fix Heading**: Denotes the state when robot heading differs from the desired heading by more than a threshold (represented by yaw_precision_ in code)
- **Go Straight**: Denotes the state when robot has **correct** heading but is away from the desired point by a distance greater than some threshold (represented by dist_precision_ in code)
- **Done**: Denotes the state when robot has correct heading and has reached the destination.

The robot can be in any one state at a time and can switch to other states as different conditions arise. This is depicted by the following state transition diagram



To implement this state logic lets create a new python script inside the **~/catkin_ws/src/two-wheeled-robot-motion-planning/scripts/** directory named **go_to_point.py** with following content

```
1   #! /usr/bin/env python
2
3   # import ros stuff
4   import rospy
5   from sensor_msgs.msg import LaserScan
6   from geometry_msgs.msg import Twist, Point
7   from nav_msgs.msg import Odometry
8   from tf import transformations
9
10  import math
11
12  # robot state variables
13  position_ = Point()
14  yaw_ = 0
15  # machine state
16  state_ = 0
17  # goal
18  desired_position_ = Point()
19  desired_position_.x = -3
20  desired_position_.y = 7
21  desired_position_.z = 0
22  # parameters
23  yaw_precision_ = math.pi / 90 # +/- 2 degree allowed
24  dist_precision_ = 0.3
25
26  # publishers
27  pub = None
28
29  # callbacks
30  def clbk_odom(msg):
31      global position_
```

```
32      global yaw_
33
34      # position
35      position_ = msg.pose.pose.position
36
37      # yaw
38      quaternion = (
39          msg.pose.pose.orientation.x,
40          msg.pose.pose.orientation.y,
41          msg.pose.pose.orientation.z,
42          msg.pose.pose.orientation.w)
43      euler = transformations.euler_from_quaternion(quaternion)
44      yaw_ = euler[2]
45
46  def change_state(state):
47      global state_
48      state_ = state
49      print 'State changed to [%s]' % state_
50
51  def fix_yaw(des_pos):
52      global yaw_, pub, yaw_precision_, state_
53      desired_yaw = math.atan2(des_pos.y - position_.y, des_pos.x - position_.x)
54      err_yaw = desired_yaw - yaw_
55
56      twist_msg = Twist()
57      if math.fabs(err_yaw) > yaw_precision_:
58          twist_msg.angular.z = 0.7 if err_yaw > 0 else -0.7
59
60      pub.publish(twist_msg)
61
62      # state change conditions
63      if math.fabs(err_yaw) <= yaw_precision_:
64          print 'Yaw error: [%s]' % err_yaw
65          change_state(1)
66
67  def go_straight_ahead(des_pos):
68      global yaw_, pub, yaw_precision_, state_
69      desired_yaw = math.atan2(des_pos.y - position_.y, des_pos.x - position_.x)
70      err_yaw = desired_yaw - yaw_
71      err_pos = math.sqrt(pow(des_pos.y - position_.y, 2) + pow(des_pos.x - position_.x, 2))
72
73      if err_pos > dist_precision_:
74          twist_msg = Twist()
75          twist_msg.linear.x = 0.6
76          pub.publish(twist_msg)
77      else:
78          print 'Position error: [%s]' % err_pos
79          change_state(2)
80
81      # state change conditions
82      if math.fabs(err_yaw) > yaw_precision_:
83          print 'Yaw error: [%s]' % err_yaw
84          change_state(0)
85
86  def done():
87      twist_msg = Twist()
88      twist_msg.linear.x = 0
89      twist_msg.angular.z = 0
```

```
90      pub.publish(twist_msg)
91
92  def main():
93      global pub
94
95      rospy.init_node('go_to_point')
96
97      pub = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
98
99      sub_odom = rospy.Subscriber('/odom', Odometry, clbk_odom)
100
101     rate = rospy.Rate(20)
102     while not rospy.is_shutdown():
103         if state_ == 0:
104             fix_yaw(desired_position_)
105         elif state_ == 1:
106             go_straight_ahead(desired_position_)
107         elif state_ == 2:
108             done()
109             pass
110         else:
111             rospy.logerr('Unknown state!')
112             pass
113         rate.sleep()
114
115 if __name__ == '__main__':
116     main()
```

Let's analyze the contents of this script. We have defined the following function:

- **main**
  This is the entry point of the file. This function sets up a **Subscriber** to the odometry topic **/odom** and a **Publisher** to **/cmd_vel** topic to command velocity of the robot. Further this function processes the state machine depending on the value of state variable.
- **clbk_odom**
  This is a callback function for the Subscriber defined in main. This function receives the odometry data and extracts the **position** and **yaw** information. The odometry data encodes orientation information in **quaternions**. To obtains **yaw** the **quaternion** is converted into **euler angles** (line 43)
- **change_state**
  This function changes the value of the global state variable that stores the robot **state** information.
- **fix_yaw**
  This function is executed when robot is in state 0 (**Fix heading**). First the **current heading** of the robot is checked with **desired heading**. If the

differene in **heading** is more than a threshold the robot is commanded to turn in its place.

- **go_straight_ahead**
  This function is executed when robot is in state 1 (**Go Straight**). This state occurs after robot has fixed the error in **yaw**. In this state, the distance between the robot's **current position** and **desired position** is compared with a threshold. If robot is further away from **desired position** it is commanded to move forward. If the **current position** lies closer to the **desired position** then the yaw is again checked for error, if yaw is significantly different from the desired yaw value the robot goes to state 0.
- **done**
  Eventually robot achieves **correct heading** and **correct position**. Once in this state the robot stops.

Step 6.2

Start a simulation using the **Simulations** menu option. Select the **world.launch** option and launch the simulation.

To test the logic lets run the simulation. We have the world loaded, now we will spawn the differential drive robot with following command

```
$ roslaunch m2wr_description spawn.launch
```

Finally, we launch the **obstacle avoidance** script to move the robot around and avoid obstacles

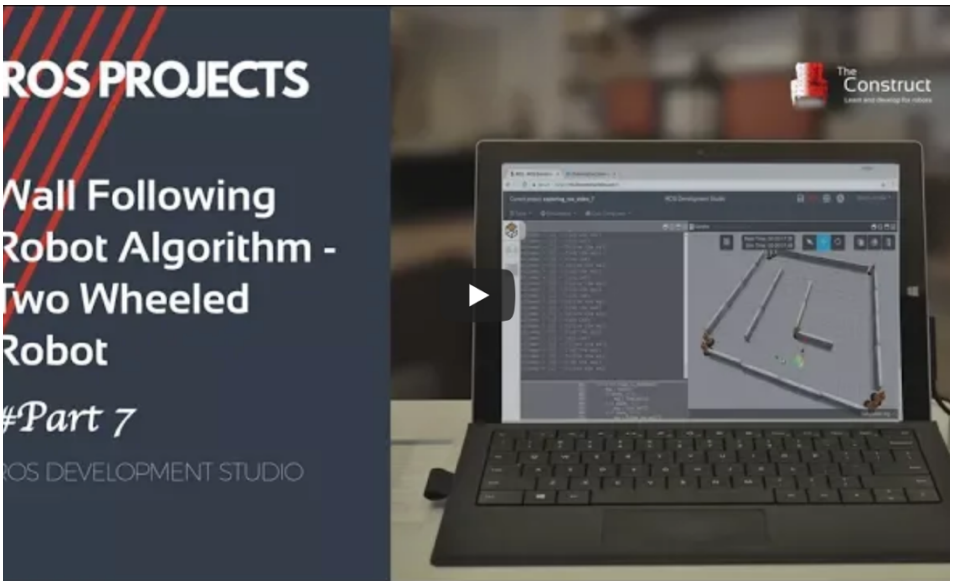
```
$ rosrun motion_plan obstacle_avoidance.py
```

Now you should see the robot navigate to the point given in the **go_to_point.py** script (line 19-21) from its current location. With the navigation implemented we have finished the part 6 of the series.

Wall Following Robot Algorithm - Two Wheeled Robot #Part 7

In this video, we are going work with wall following robot algorithm. We'll start watching the demo, then let's go straight to the code and understand line by line how to perform the task.

Video Tutorial Part 7



Steps to recreate the project as shown in the video

Step 7.1

In this part we will write an algorithm to make the robot follow a wall. We can continue from the last part or start with a new project. These instructions are for starting with a new project.

- Go to ROS Development Studio and create a new project.
- Provide a suitable project name and some useful description. (We have named the project **exploring_ros_video_7**)

- Load/Start the project.

Now We will fetch the project files we have developed in previous part:

- Clone the *bitbucket* repository two-wheeled-robot - Simulation.
- Checkout to the **right** branch (here is more information about branch and branching in git)
- Open **Tools > Shell** and run the following commands

```
$ cd simulation_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-
wheeled-robot-simulation.git
$ cd two-wheeled-robot-simulation
$ git checkout 16e45ce
```

- Compile the project to make it ready to use

```
cd ~/simulation_ws/src
catkin_make
```

- Clone another github repository two-wheeled-robot - Motion Planning. Open **Tools > Shell** and run the following commands

```
$ cd catkin_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-
wheeled-robot-motion-planning.git
$ cd two-wheeled-robot-motion-planning
$ git checkout f62dda2
```

- Compile the project to make it ready to use

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

- At this point we should have the following directory structure

```
.
├── ai_ws
├── catkin_ws
|   ├── build
|   ├── devel
|   └── src
|       ├── CMakeLists.txt
|       └── two-wheeled-robot-motion-planning
|           ├── CMakeLists.txt
|           ├── package.xml
|           └── scripts
|               ├── follow_wall.py
|               ├── go_to_point.py
|               ├── obstacle_avoidance.py
|               └── reading_laser.py
├── explore_ros_video_7.zip
├── notebook_ws
|   ├── default.ipynb
|   └── images
└── simulation_ws
    ├── build
    ├── devel
    └── src
        ├── CMakeLists.txt
        └── two-wheeled-robot-simulation
            ├── m2wr_description
            |   ├── CMakeLists.txt
            |   ├── launch
            |   ├── package.xml
            |   └── urdf
            └── my_worlds
                ├── CMakeLists.txt
                ├── launch
                ├── package.xml
                └── worlds
```

Step 7.2

- Launch the simulation with **Simulations > Select launch file > my_worlds > world.launch**
- Spawn the robot with following command

```
$ roslaunch m2wr_description spawn.launch
```

- The wall following algorithm is written inside the **follow_wall.py** script located within the **~catkin_ws/src/two-wheeled-robot-motion-**

planning/scripts/ directory. Open **Tools>IDE** and browse to this script. It contains following code

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

import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf import transformations

import math

pub_ = None
regions_ = {
        'right': 0,
        'fright': 0,
        'front': 0,
        'fleft': 0,
        'left': 0,
}
state_ = 0
state_dict_ = {
    0: 'find the wall',
    1: 'turn left',
    2: 'follow the wall',
}

def clbk_laser(msg):
    global regions_
    regions_ = {
        'right':  min(min(msg.ranges[0:143]), 10),
        'fright': min(min(msg.ranges[144:287]), 10),
        'front':  min(min(msg.ranges[288:431]), 10),
        'fleft':  min(min(msg.ranges[432:575]), 10),
        'left':   min(min(msg.ranges[576:713]), 10),
    }

    take_action()

def change_state(state):
    global state_, state_dict_
    if state is not state_:
        print 'Wall follower - [%s] - %s' % (state, state_dict_[state])
        state_ = state

def take_action():
    global regions_
    regions = regions_
    msg = Twist()
    linear_x = 0
    angular_z = 0
        state_description = ''

    d = 1.5

```

```
54      if regions['front'] > d and regions['fleft'] > d and regions['fright'] > d:
55          state_description = 'case 1 - nothing'
56          change_state(0)
57      elif regions['front'] < d and regions['fleft'] > d and regions['fright'] > d:
58          state_description = 'case 2 - front'
59          change_state(1)
60      elif regions['front'] > d and regions['fleft'] > d and regions['fright'] < d:
61          state_description = 'case 3 - fright'
62          change_state(2)
63      elif regions['front'] > d and regions['fleft'] < d and regions['fright'] > d:
64          state_description = 'case 4 - fleft'
65          change_state(0)
66      elif regions['front'] < d and regions['fleft'] > d and regions['fright'] < d:
67          state_description = 'case 5 - front and fright'
68          change_state(1)
69      elif regions['front'] < d and regions['fleft'] < d and regions['fright'] > d:
70          state_description = 'case 6 - front and fleft'
71          change_state(1)
72      elif regions['front'] < d and regions['fleft'] < d and regions['fright'] < d:
73          state_description = 'case 7 - front and fleft and fright'
74          change_state(1)
75      elif regions['front'] > d and regions['fleft'] < d and regions['fright'] < d:
76          state_description = 'case 8 - fleft and fright'
77          change_state(0)
78      else:
79          state_description = 'unknown case'
80          rospy.loginfo(regions)
81
82  def find_wall():
83      msg = Twist()
84      msg.linear.x = 0.2
85      msg.angular.z = -0.3
86      return msg
87
88  def turn_left():
89      msg = Twist()
90      msg.angular.z = 0.3
91      return msg
92
93  def follow_the_wall():
94      global regions_
95
96      msg = Twist()
97      msg.linear.x = 0.5
98      return msg
99
100 def main():
101     global pub_
102
103     rospy.init_node('reading_laser')
104
105     pub_ = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
106
107     sub = rospy.Subscriber('/m2wr/laser/scan', LaserScan, clbk_laser)
108
109     rate = rospy.Rate(20)
110     while not rospy.is_shutdown():
111         msg = Twist()
```

```
112         if state_ == 0:
113             msg = find_wall()
114         elif state_ == 1:
115             msg = turn_left()
116         elif state_ == 2:
117             msg = follow_the_wall()
118             pass
119         else:
120             rospy.logerr('Unknown state!')
121 
122         pub_.publish(msg)
123 
124         rate.sleep()
125 
126 if __name__ == '__main__':
127     main()
```

Let's analyze the contents of this scripts:

- There are a few global variables
  - pub_: this variable is a reference to the publisher that will be initialized inside the main function
  - regions_: this is a dictionary variable that holds the distances to five directions
  - state_: this variable stores the current state of the robot
  - state_dict_: this variable holds the possible states

- The functions defined are:
  - main: This is the entry point for the algorithm, it initializes a node, a publisher and a subscriber. Depending on the value of the state_ variable, a suitable control action is taken (by calling other functions). This function also configures the frequency of execution of control action using Rate function.
  - clbk_laser: This function is passed to the Subscriber method and it executes when a new laser data is made available. This function writes **distance** values in the global variable regions_ and calls the function take_actions
  - take_action: This function manipulates the **state** of the robot. The various distances stored in regions_ variable help in determining the **state** of the robot.
  - find_wall: This function defines the action to be taken by the robot when it is not surrounded by any obstacle. This method essentially makes the robot move in an anti-clockwise circle (until it finds a wall).

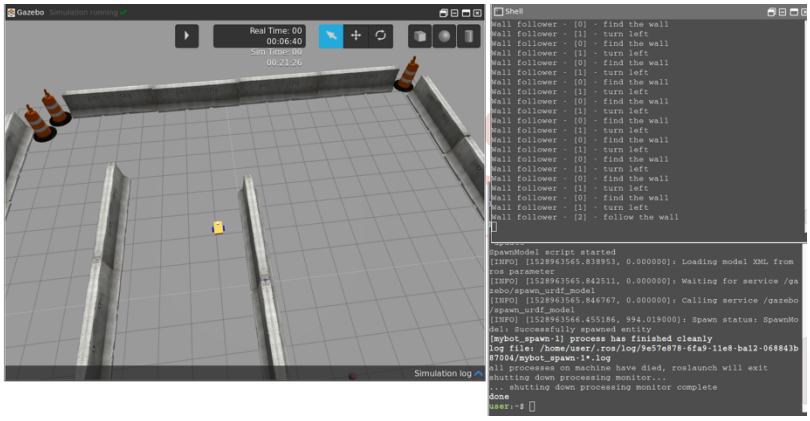
- turn_left: When the robot detects an obstacle it executes the turn left action
- follow_the_wall: Once the robot is positioned such that its front and front-left path is clear while its front-right is obstructed the robot goes into the **follow wall** state. In this state this function is executed and this function makes the robot to follow a straight line.

This is the overall logic that governs the **wall following** behavior of the robot.

Step 7.3

Let us run the **wall following** algorithm and see the robot in action. We already have the simulation window open with robot spawned into it. Open **Tools>Shell** and enter the following command

```
$ rosrun motion_plan follow_wall
```



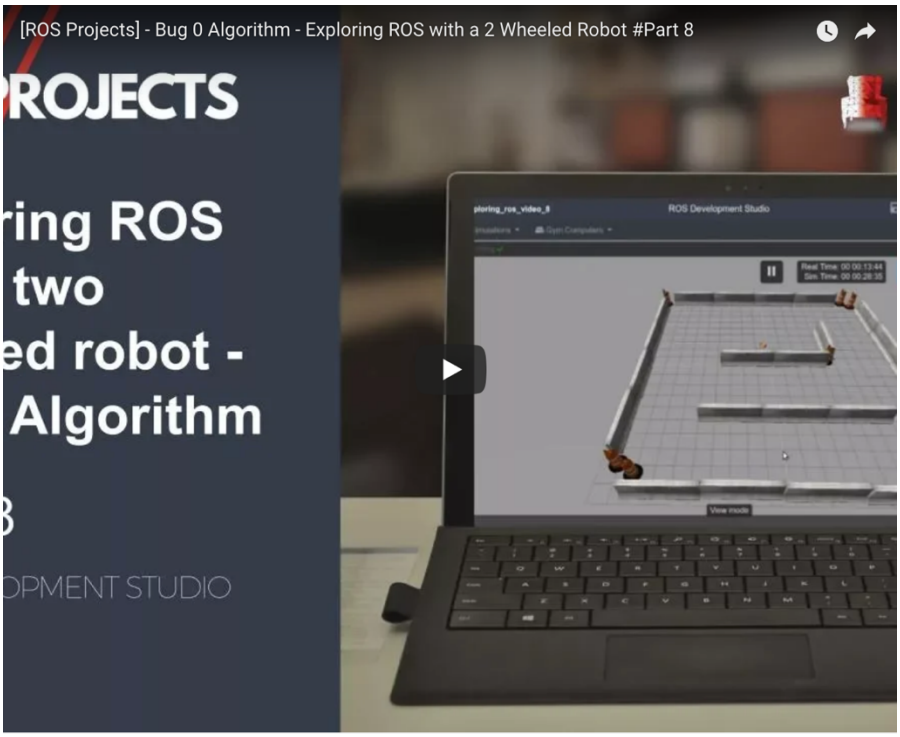
This finishes the part 7.

REFERENCES
RDS: HTTPS://RDS.THECONSTRUCTSIM.COM/
SIMULATION: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT
MOTION PACKAGE: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT-MOTION-PLANNING

Bug 0 Algorithm - Two Wheeled Robot #Part 8

In this video, the 8th of the series Exploring ROS with a 2 Wheeled Robot, we are going to work with the Bug 0 algorithm using the previous scripts we have created before: Wall following + Go to point

Video Tutorial Part 8



Steps to recreate the project as shown in the video

Step 8.1

In this part we are going to program **Bug 0** behavior for our mobile robot. Basic requirements of **Bug 0** algorithm are:

- direction to goal should be known
- Wall sensing ability

We will start by creating a new project and cloning the project files in our project.

- Go to ROS Development Studio and create a new project.
- Provide a suitable project name and some useful description. (We have named the project **exploring_ros_video_8**)
- Load/Start the project.

Now We will fetch the project files we have developed in previous part:

- Clone the *bitbucket* repository two-wheeled-robot - Simulation.
- Checkout to the **right** branch (here is more information about branch and branching in git)
- Open **Tools > Shell** and run the following commands

```
$ cd simulation_ws/src
$ git clone https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-simulation.git
$ cd two-wheeled-robot-simulation
$ git checkout 16e45ce
```

- Compile the project to make it ready to use

```
cd ~/simulation_ws/src
catkin_make
```

- Clone another github repository two-wheeled-robot - Motion Planning. Open **Tools > Shell** and run the following commands

```
$ cd catkin_ws/src
$ git clone https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning.git
$ cd two-wheeled-robot-motion-planning $ git checkout be714ee
```

- Compile the project to make it ready to use

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

Step 8.2

The script **bug0.py** inside the **catkin_ws/src/two-wheeled-robot-motion-planning/scripts/** implements the **Bug 0** behavior. In short, the **Bug 0** algorithm drives the robot towards a points (goal), while doing so if the robot detects an obstacle it goes around it.

The important functions we have defines in the **bug0.py** script are:

- main: The entry point of the program. It initializes a node, two subscribers (laser scan and odometry) and two **Service clients** (go_to_point_switch and wall_follower_switch). A **state** based logic is used to drive robot towards the **goal** position. Initially the robot is put in Go to point state, and when an obstacle is detected the state is switched to Wall following state. When there is a straight free path towards the goal the robot again switches to state Go to point.
- change_state: This function accepts a state argument and calls the respective service handler.

Other functions are similar to those implemented before in part 7 (eg. clbk_odom, clbk_laser and normalize_angle).

Also we have done changes to previously created scripts **follow_wall.py** and **go_to_point.py**. These scripts now implement an additional **service** server. The server helps in *activating* and *deactivating* the execution of respective algorithm based on the **state** maintained in the **bug0.py** script.

For example, when we are in Go to point state we communicate the server (go_to_point_switch) to activate go_to_point algorithm and we deactivate the other server (wall_follower_switch).

Here is the code for **bug0.py** script for reference:

```
1   #! /usr/bin/env python
2
3   import rospy
4   # import ros message
```

```
5   from geometry_msgs.msg import Point
6   from sensor_msgs.msg import LaserScan
7   from nav_msgs.msg import Odometry
8   from tf import transformations
9   # import ros service
10  from std_srvs.srv import *
11
12  import math
13
14  srv_client_go_to_point_ = None
15  srv_client_wall_follower_ = None
16  yaw_ = 0
17  yaw_error_allowed_ = 5 * (math.pi / 180) # 5 degrees
18  position_ = Point()
19  desired_position_ = Point()
20  desired_position_.x = rospy.get_param('des_pos_x')
21  desired_position_.y = rospy.get_param('des_pos_y')
22  desired_position_.z = 0
23  regions_ = None
24  state_desc_ = ['Go to point', 'wall following']
25  state_ = 0
26  # 0 - go to point
27  # 1 - wall following
28
29  # callbacks
30  def clbk_odom(msg):
31      global position_, yaw_
32
33      # position
34      position_ = msg.pose.pose.position
35
36      # yaw
37      quaternion = (
38          msg.pose.pose.orientation.x,
39          msg.pose.pose.orientation.y,
40          msg.pose.pose.orientation.z,
41          msg.pose.pose.orientation.w)
42      euler = transformations.euler_from_quaternion(quaternion)
43      yaw_ = euler[2]
44
45  def clbk_laser(msg):
46      global regions_
47      regions_ = {
48          'right':  min(min(msg.ranges[0:143]), 10),
49          'fright': min(min(msg.ranges[144:287]), 10),
50          'front':  min(min(msg.ranges[288:431]), 10),
51          'fleft':  min(min(msg.ranges[432:575]), 10),
52          'left':   min(min(msg.ranges[576:719]), 10),
53      }
54
55  def change_state(state):
56      global state_, state_desc_
57      global srv_client_wall_follower_, srv_client_go_to_point_
58      state_ = state
59      log = "state changed: %s" % state_desc_[state]
60      rospy.loginfo(log)
61      if state_ == 0:
62          resp = srv_client_go_to_point_(True)
```

```
63          resp = srv_client_wall_follower_(False)
64      if state_ == 1:
65          resp = srv_client_go_to_point_(False)
66          resp = srv_client_wall_follower_(True)
67
68
69  def normalize_angle(angle):
70      if(math.fabs(angle) > math.pi):
71          angle = angle - (2 * math.pi * angle) / (math.fabs(angle))
72      return angle
73
74  def main():
75      global regions_, position_, desired_position_, state_, yaw_, yaw_error_allowed_
76      global srv_client_go_to_point_, srv_client_wall_follower_
77
78      rospy.init_node('bug0')
79
80      sub_laser = rospy.Subscriber('/m2wr/laser/scan', LaserScan, clbk_laser)
81      sub_odom = rospy.Subscriber('/odom', Odometry, clbk_odom)
82
83      srv_client_go_to_point_ = rospy.ServiceProxy('/go_to_point_switch', SetBool)
84      srv_client_wall_follower_ = rospy.ServiceProxy('/wall_follower_switch', SetBool)
85
86      # initialize going to the point
87      change_state(0)
88
89      rate = rospy.Rate(20)
90      while not rospy.is_shutdown():
91          if regions_ == None:
92              continue
93
94          if state_ == 0:
95              if regions_['front'] > 0.15 and regions_['front'] < 1:
96                  change_state(1)
97
98          elif state_ == 1:
99              desired_yaw = math.atan2(desired_position_.y - position_.y, desired_position_.x - position_.x)
100             err_yaw = normalize_angle(desired_yaw - yaw_)
101
102             if math.fabs(err_yaw) < (math.pi / 6) and \
103               regions_['front'] > 1.5:
104                change_state(0)
105
106             if err_yaw > 0 and \
107               math.fabs(err_yaw) > (math.pi / 4) and \
108               math.fabs(err_yaw) < (math.pi / 2) and \
109               regions_['left'] > 1.5:
110                change_state(0)
111
112             if err_yaw < 0 and \
113               math.fabs(err_yaw) > (math.pi / 4) and \
114               math.fabs(err_yaw) < (math.pi / 2) and \
115               regions_['right'] > 1.5:
116                change_state(0)
117
118         rate.sleep()
119
120 if __name__ == "__main__":
```

```
121    main()
```

Lastly, we have defined a new launch file (inside directory ~catkin_ws/src/two-wheeled-robot-motion-planning/launch) that will help us run all these scripts. We can manually run individual scripts too but that is tiresome. The script behaviors.launch helps us launch the two behaviors i.e. wall follower and go to point. Here are the contents this launch file for reference

```
<launch>
   <arg name="des_x" />
   <arg name="des_y" />
   <param name="des_pos_x" value="$(arg des_x)" />
   <param name="des_pos_y" value="$(arg des_y)" />
   <node pkg="motion_plan" type="follow_wall.py" name="wall_follower" output="screen" />
   <node pkg="motion_plan" type="go_to_point.py" name="go_to_point" output="screen" />
</launch>
```

Step 8.3

Let us now run the simulation and see the robot in action

- Launch the simulation with **Simulations > Select launch file > my_worlds > world.launch**
- Spawn the robot with following command

```
$ roslaunch m2wr_description spawn.launch
```

- Launch the behavior nodes

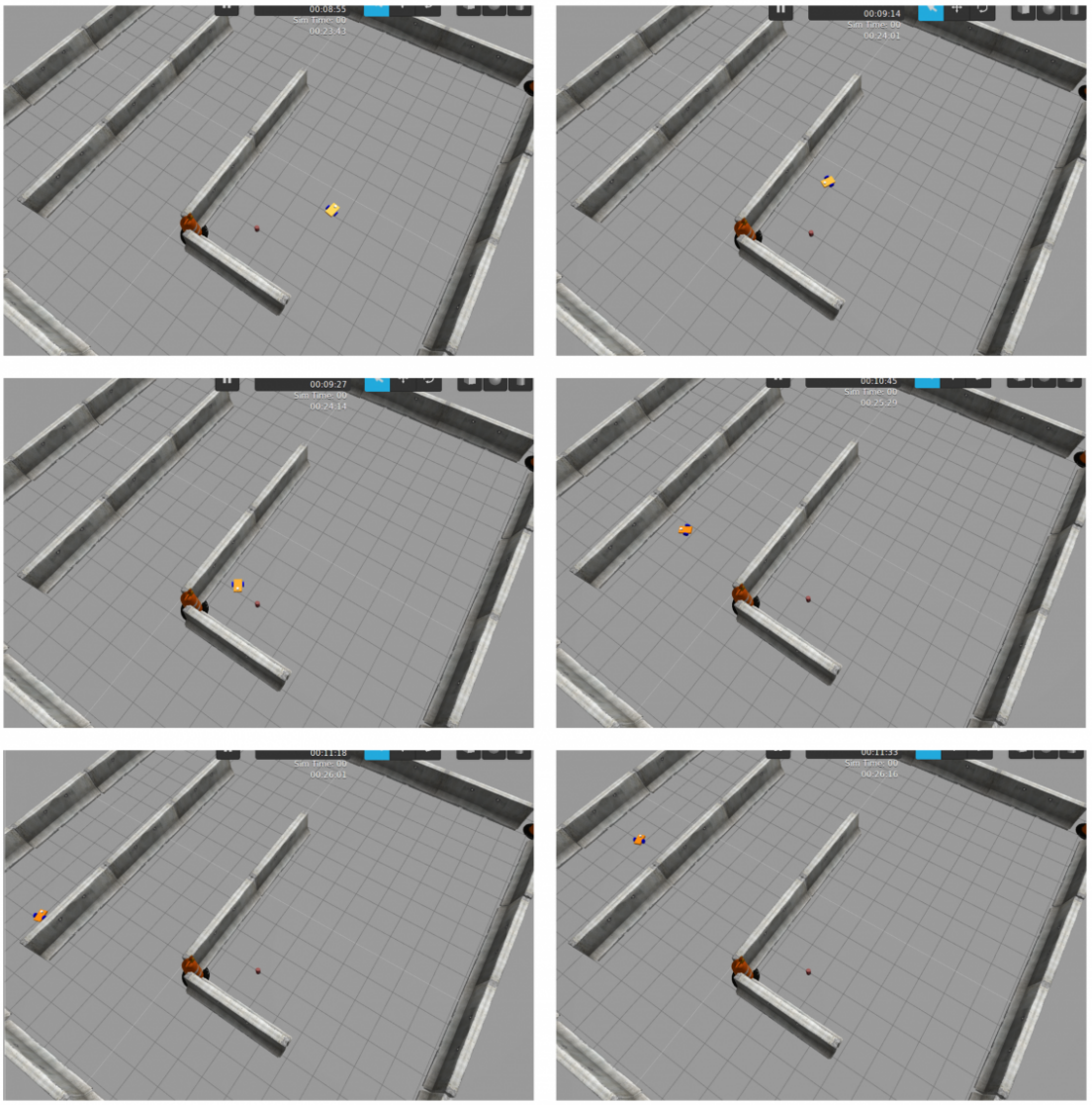
```
$ roslaunch motion_plan behaviors.launch des_x:=0 des_y:=8
```

Notice the last two arguments are the co-ordinates of the **goal** location

- Run the bug0 algorithm

```
$ rosrun motion_plan bug0.py
```

Now the robot should navigate towards the goal location.



References

RDS: https://rds.theconstructsim.com/

Simulation: https://bitbucket.org/theconstructcore/two-wheeled-robot

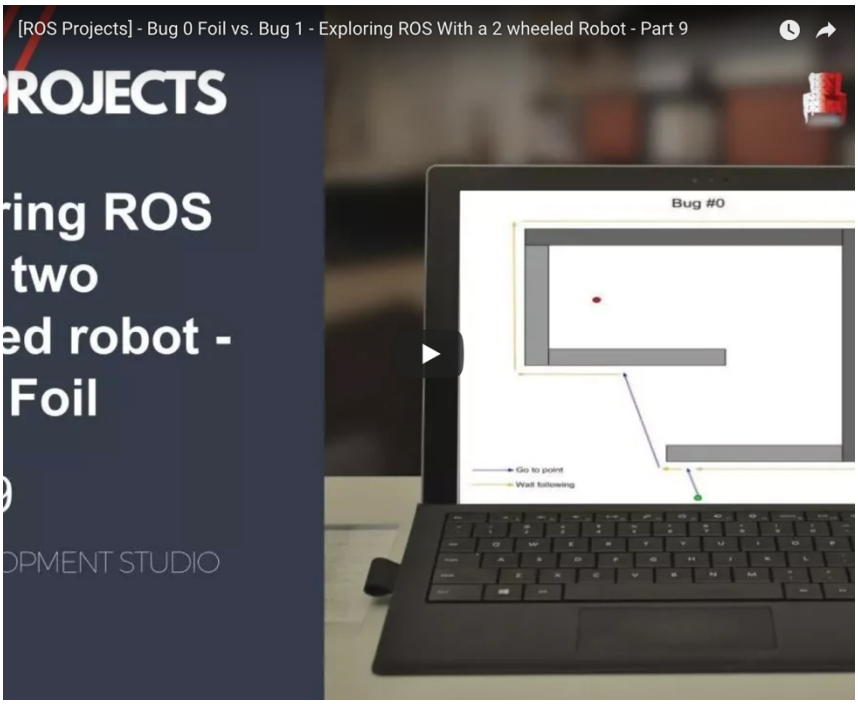
Motion package: https://bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning

ROS Basics: Robot Ignite Academy

Bug 0 Foil vs. Bug 1 - Two Wheeled Robot #Part 9

In this video, the 9th of the series Exploring ROS with a 2 Wheeled Robot, we are going to see the Bug 0 Foil, why it happens and how it is improved using the Bug 1 behavior.

Video Tutorial Part 9



Below are the steps to replicate the project as shown in the video

Step 9.1

We will start by creating a new project and cloning the project files in our project.

- Go to ROS Development Studio and create a new project.
- Provide a suitable project name and some useful description. (We have named the project **exploring_ros_video_9**)
- Load/Start the project.

Now We will fetch the project files we have developed in previous part:

- Clone the *bitbucket* repository two-wheeled-robot - Simulation.
- Checkout to the **right** branch (here is more information about branch and branching in git)
- Open **Tools > Shell** and run the following commands

```
$ cd simulation_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-simulation.git
$ cd two-wheeled-robot-simulation
$ git checkout 0d263ae
```

- Compile the project to make it ready to use

```
cd ~/simulation_ws/src
catkin_make
```

- Clone another github repository two-wheeled-robot - Motion Planning. Open **Tools > Shell** and run the following commands

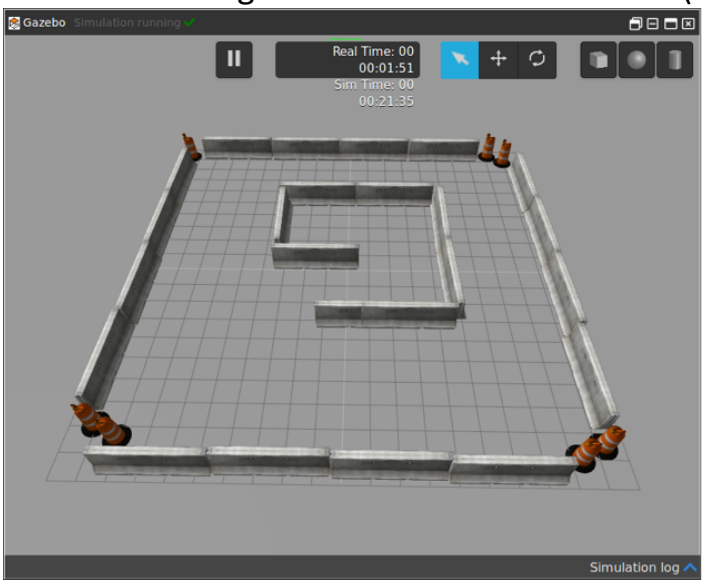
```
$ cd catkin_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning.git
$ cd two-wheeled-robot-motion-planning $ git checkout 1b69124
```

- Compile the project to make it ready to use

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

Step 9.2

- Launch the new simulation world with **Simulations > Select launch file > world.launch** This will start the gazebo window with a world (as shown)



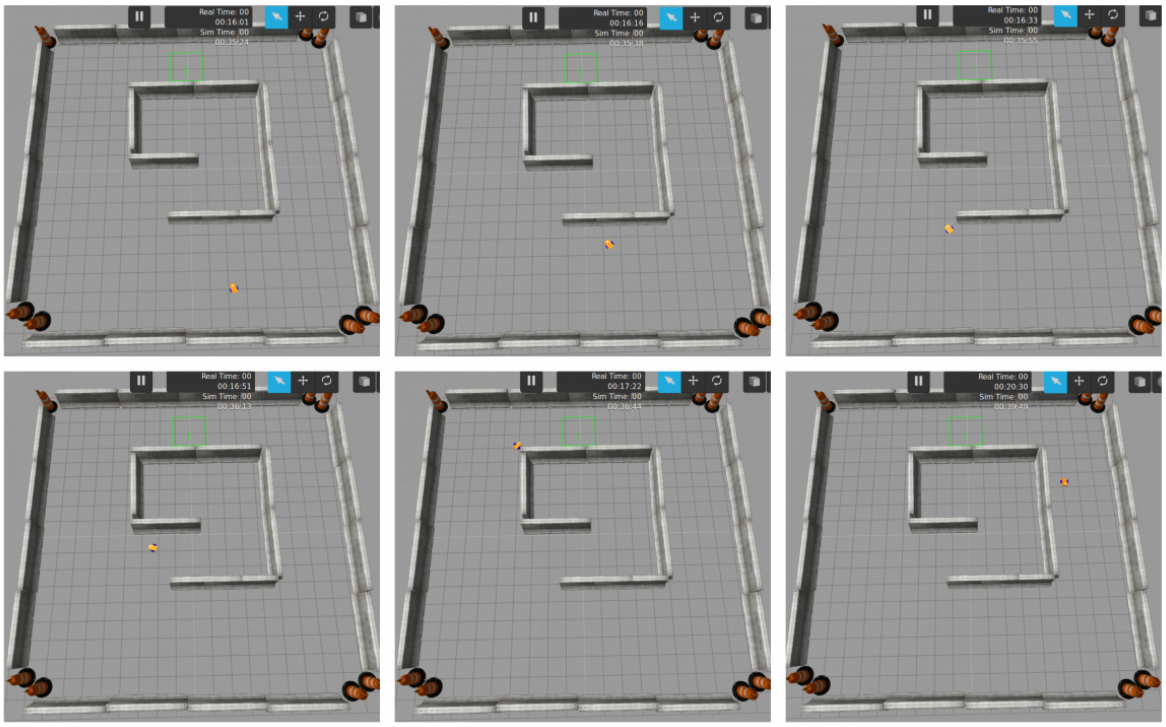
- Open **Tools > Shell** and spawn the robot with following commands on shell

```
$ roslaunch m2wr_description spawn.launch y:=8
```

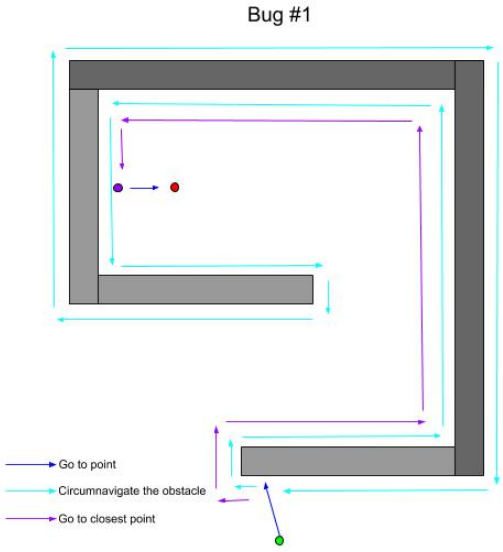
- Once the robot is loaded into the world. Let's run our **Bug 0** algorithm to navigate to a point **x = 2** and **y = -3**

```
$ roslaunch motion_plan bug0.launch des_x:=2 des_y:=-3
```

Notice that the robot moves in the circumference of the new fencing structure in the world, never reaching the goal position (as shown)



Thus we see the inherent drawback of the **Bug 0** algorithm. We can do better using the **Bug 1** algorithm as depicted in the image below



The **Bug 1** algorithm moves the robot about the obstacle (circumnavigate). When the robot passes near the goal it records this point and keeps on circumnavigating the obstacle. Once the robot reaches the initial point (where the robot first met the obstacle) it then goes to the point stored in memory and then moves towards the goal from there.

We can see that **Bug 1** algorithm, though lengthy, works better in situations where **Bug 0** will fail.

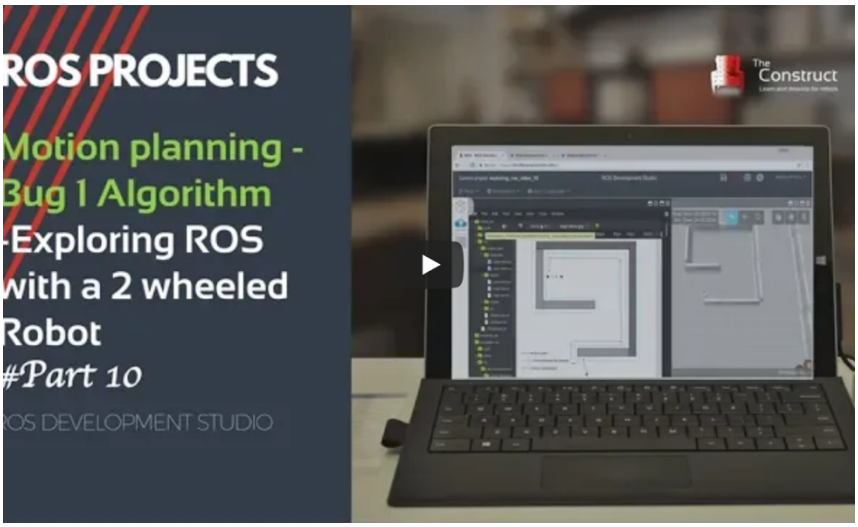
In the next part we will implement the **Bug 1** algorithm. This finishes the part 9.

References
RDS: ROS Development Studio
Simulation: https://bitbucket.org/theconstructcore/two-wheeled-robot
Motion package: https://bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning
ROS Basics: Robot Ignite Academy

Bug 1 - Two Wheeled Robot #Part 10

In this video an algorithm to perform the motion planning task Bug 1 is implemented using a machine state.

Video Tutorial Part10



Steps to replicate the project as shown in the video

Step 10.1

We will start by creating a new project and cloning the project files in our project.

- Go to ROS Development Studio and create a new project.
- Provide a suitable project name and some useful description. (We have named the project **exploring_ros_video_10**)
- Load/Start the project.

Now We will fetch the project files we have developed in previous part:

- Clone the *bitbucket* repository two-wheeled-robot - Simulation.
- Checkout to the **right** branch (here is more information about branch and branching in git)
- Open **Tools > Shell** and run the following commands

```
$ cd simulation_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-simulation.git
$ cd two-wheeled-robot-simulation
$ git checkout 0d263ae
```

- Compile the project to make it ready to use

```
cd ~/simulation_ws/src
catkin_make
```

- Clone another github repository two-wheeled-robot - Motion Planning. Open **Tools > Shell** and run the following commands

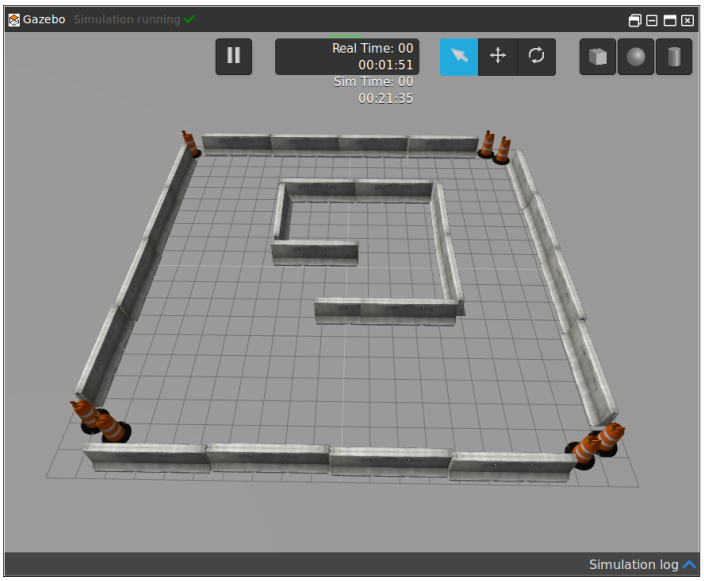
```
$ cd catkin_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning.git
$ cd two-wheeled-robot-motion-planning
$ git checkout 300b107
```

- Compile the project to make it ready to use

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

Step 10.2

- Load the world file **Simulations > Select launch file > world.launch**



- Let's spawn the robot at **x=0, y=8**. Open **Tools > Shell** and enter the following commands

```
$ roslaunch m2wr_description spawn.launch y:=8
```

- Next we set the **goal** (point **x=0, y=-3**) for our robot and launch the **Bug 1** behavior with following command

```
$ roslaunch motion_plan bug1.launch des_x:=0 des_y:=-3
```

- While the robot performs the navigation, we can analyze the structure and contents of **Bug 1** algorithm

Here is the code for the bug1.py script contained in ~/catkin_ws/src/two-wheeled-robot-motion-planning/scripts/ directory:

```
import rospy
# import ros message
from geometry_msgs.msg import Point
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from tf import transformations
# import ros service
from std_srvs.srv import *

import math

srv_client_go_to_point_ = None
srv_client_wall_follower_ = None
yaw_ = 0
yaw_error_allowed_ = 5 * (math.pi / 180) # 5 degrees
position_ = Point()
desired_position_ = Point()
desired_position_.x = rospy.get_param('des_pos_x')
desired_position_.y = rospy.get_param('des_pos_y')
desired_position_.z = 0
regions_ = None
state_desc_ = ['Go to point', 'circumnavigate obstacle', 'go to closest point']
state_ = 0
circumnavigate_starting_point_ = Point()
circumnavigate_closest_point   = Point()
count_state_time_ = 0 # seconds the robot is in a state
count_loop_ = 0
# 0 - go to point
# 1 - circumnavigate
# 2 - go to closest point

# callbacks
def clbk_odom(msg):
    global position_, yaw_

    # position
    position_ = msg.pose.pose.position

    # yaw
    quaternion = (
        msg.pose.pose.orientation.x,
        msg.pose.pose.orientation.y,
        msg.pose.pose.orientation.z,
        msg.pose.pose.orientation.w)
    euler = transformations.euler_from_quaternion(quaternion)
    yaw_ = euler[2]

def clbk_laser(msg):
    global regions_
    regions_ = {
        'right':  min(min(msg.ranges[0:143]), 10),
        'fright': min(min(msg.ranges[144:287]), 10),
        'front':  min(min(msg.ranges[288:431]), 10),
        'fleft':  min(min(msg.ranges[432:575]), 10),
        'left':   min(min(msg.ranges[576:719]), 10),
    }

```

```
58 def change_state(state):
59     global state_, state_desc_
60     global srv_client_wall_follower_, srv_client_go_to_point_
61     global count_state_time_
62     count_state_time_ = 0
63     state_ = state
64     log = "state changed: %s" % state_desc_[state]
65     rospy.loginfo(log)
66     if state_ == 0:
67         resp = srv_client_go_to_point_(True)
68         resp = srv_client_wall_follower_(False)
69     if state_ == 1:
70         resp = srv_client_go_to_point_(False)
71         resp = srv_client_wall_follower_(True)
72     if state_ == 2:
73         resp = srv_client_go_to_point_(False)
74         resp = srv_client  wall  follower  (True)
75 
76 def calc_dist_points(point1, point2):
77     dist = math.sqrt((point1.y - point2.y)**2 + (point1.x - point2.x)**2)
78     return dist
79 
80 def normalize_angle(angle):
81     if(math.fabs(angle) > math.pi):
82         angle = angle - (2 * math.pi * angle) / (math.fabs(angle))
83     return angle
84 
85 def main():
86     global regions_, position_, desired_position_, state_, yaw_, yaw_error_allowed_
87     global srv_client_go_to_point_, srv_client_wall_follower_
88     global circumnavigate_closest_point_, circumnavigate_starting_point_
89     global count_loop_, count_state_time_
90 
91     rospy.init_node('bug1')
92 
93     sub_laser = rospy.Subscriber('/m2wr/laser/scan', LaserScan, clbk  laser)
94     sub_odom = rospy.Subscriber('/odom', Odometry, clbk_odom)
95 
96     rospy.wait_for_service('/go_to_point_switch')
97     rospy.wait_for_service('/wall_follower_switch')
98 
99     srv_client_go_to_point   = rospy.ServiceProxy('/go  to  point  switch', SetBool)
100    srv_client_wall_follower_ = rospy.ServiceProxy('/wall_follower_switch', SetBool)
101 
102    # initialize going to the point
103    change_state(0)
104 
105    rate_hz = 20
106    rate = rospy.Rate(rate_hz)
107    while not rospy.is_shutdown():
108        if regions_ == None:
109            continue
110 
111        if state_ == 0:
112            if regions_['front'] > 0.15 and regions_['front'] < 1:
113                circumnavigate_closest_point_ = position_
114                circumnavigate_starting_point_ = position_
115                change_state(1)
116 
117        elif state_ == 1:
118            # if current position is closer to the goal than the previous closest_position, assign current position to closest_point
119 
```

```
120             if calc_dist_points(position_, desired_position_) < calc_dist_points(circumnavigate_closest_point_,
121 desired_position_):
122                 circumnavigate_closest_point_ = position_
123
124             # compare only after 5 seconds - need some time to get out of starting_point
125             # if robot reaches (is close to) starting point
126             if count_state_time_ > 5 and \
127                calc_dist_points(position_, circumnavigate_starting_point_) < 0.2:
128                 change_state(2)
129
130         elif state_ == 2:
131             # if robot reaches (is close to) closest point
132             if calc_dist_points(position_, circumnavigate_closest_point_) < 0.2:
133                 change_state(0)
134
135         count_loop_ = count_loop_ + 1
136         if count_loop_ == 20:
137             count_state_time_ = count_state_time_ + 1
138             count_loop_ = 0
139
140         rate.sleep()
141
    if __name__ == "__main    ":
        main()
```

Following are the important changes:

- Addition of new states in the robot states. These states are circumnavigating obstacle and go to closest point, as discussed in the previous part, the first one makes the robot go around the obstacle perimeter and the second state makes the robot to navigate to the point (on obstacle perimeter) that takes the robot closest to the goal.

- We have new variables to store the start point and the closest point.

- The change state function is similar to that in **Bug 0** with more number of states (3):
  - State 1: Go to point: Uses the **go_to_point** algorithm
  - State 2: circumnavigate obstacle: Uses the **follow_wall** algorithm
  - State 3: go to closest point: Uses the **follow_wall** algorithm

  In the last two states, we are using the same algorithm but there is a difference of the stop point. For the state 2, the stop point is the start point (circumnavigate) while for the state 3 the stop point is the closest point

- A new function calc_dist_points is defined for calculating the distance of **goal** point from the current location of the robot.

- Lastly, there is an additional variable that stores the number of seconds elapsed while in a state. This variable helps us in determining if we have completed the circumnavigation otherwise there can be ambiguity when the robot has to change from state 2 to state 3.

This finishes the Bug 1 algorithm.

References

RDS: ROS Development Studio
Simulation: https://bitbucket.org/theconstructcore/two-wheeled-robot
Motion package: https://bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning
ROS Basics: Robot Ignite Academy

From ROS Indigo to Kinetic - Exploring ROS with a 2 wheeled Robot - Part 11

In this video, the 11th of the series, you'll see the necessary changes in the project to make it work with ROS Kinetic, the new version supported by RDS (ROS Development Studio).

Video Tutorial Part 11

Steps to recreate the project as shown in the video

Step 11.1

Lets start by creating a new project on RDS and cloning the project files from online repository.

- Go to ROS Development Studio and create a new project.
- Provide a suitable project name and some useful description. (We have named the project **exploring_ros_video_11**)
- Load/Start the project.

Now We will fetch the project files we have developed in previous part:

- Clone the *bitbucket* repository two-wheeled-robot - Simulation.
- Checkout to the **right** branch (here is more information about branch and branching in git)
- Open **Tools > Shell** and run the following commands

```
$ cd simulation_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-simulation.git
```

- Compile the project to make it ready to use

```
cd ~/simulation_ws/src
catkin_make
```

- Clone another github repository two-wheeled-robot - Motion Planning. Open **Tools > Shell** and run the following commands

```
$ cd catkin_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning.git
$ cd two-wheeled-robot-motion-planning
$ git checkout 3c130c8
```

- Compile the project to make it ready to use

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

Step 11.2

At the time of creation of these video tutorials, RDS got upgraded from **ROS Indigo** to **ROS Kinetic**. While the code we have written in previous parts works fine, we can improve upon the organization and improve usability of the code. Thus, we have made the following changes to the project since the last part.

- **Removed** Legacy mode from the differential drive plugin (otherwise we see warning)

- Some minor modification to the launch file. Earlier we had the following syntax to import a file<param name="robot_description" command="$(find xacro)xacro.py '$(find mono_bot)/urdf/mono_b.xacro'" />
  Now we need not write the **.py** extension and also need to add **--inorder** argument <param name="robot_description" command="$(find xacro)xacro --inorder '$(find mono_bot)/urdf/mono_b.xacro'" />

- Next the macro tags are also fixed in **macro.xacro** file<xacro name="link_wheel" params="name"> changes to <xacro:macro name="link_wheel" params="name">

- Lastly the opening element (robot tag) needs a modification too<robot> will change to <robot xmlns:xacro="https://www.ros.org/wiki/xacro">

Next, we will integrate the **robot spawning** code into the **simulation launch** file. This will make the job of starting a simulation easier and faster as now the robot will automatically get spawned in a desired location when simulation starts.

Here is the launch file code (**bug1.launch**) for reference

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <launch>
4   <arg name="robot" default="machines"/>
5   <arg name="debug" default="false"/>
6   <arg name="gui" default="true"/>
7   <arg name="headless" default="false"/>
8   <arg name="pause" default="false"/>
```

```
  <arg name="world" default="world03" />
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find my_worlds)/worlds/$(arg world).world"/>
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg pause)"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="headless" value="$(arg headless)"/>
    <env name="GAZEBO_MODEL_PATH" value="$(find simulation_gazebo)/models:$(optenv
GAZEBO_MODEL_PATH)"/>
  </include>
  <include file="$(find m2wr_description)/launch/spawn.launch">
      <arg name="y" value="8" />
  </include>
<!--  Include launch.xml if needed -->
</launch>
```

Finally, we will create a script to do a robot position reset because every time we make some change to our algorithm we need to start the simulation again and again. With the help of script, we will not need to restart the simulation. This is possible because **gazebo** provides a node called **/gazebo/set_model_state** to set the model state. The following script shows how this is done

```
<span class="cm-keyword">import</span> <span class="cm-variable">rospy</span>
<span class="cm-keyword">from</span> <span class="cm-variable">gazebo_msgs</span>.<span class="cm-
property">srv</span> <span class="cm-keyword">import</span> <span class="cm-variable">SetModelState</span>
<span class="cm-keyword">from</span> <span class="cm-variable">gazebo_msgs</span>.<span class="cm-
property">msg</span> <span class="cm-keyword">import</span> <span class="cm-variable">ModelState</span>

<span class="cm-variable">srv_client_set_model_state</span> = <span class="cm-variable">rospy</span>.<span
class="cm-property">ServiceProxy</span>(<span class="cm-string">'/gazebo/set_model_state'</span>, <span class="cm-
variable">SetModelState</span>)
<span class="cm-variable">model  state</span> = <span class="cm-variable">ModelState</span>()
<span class="cm-variable">model_state</span>.<span class="cm-property">model_name</span> = <span class="cm-
string">'m2wr'</span>
<span class="cm-variable">model_state</span>.<span class="cm-property">pose</span>.<span class="cm-
property">position</span>.<span class="cm-property">x</span> = <span class="cm-number">0</span>
<span class="cm-variable">model_state</span>.<span class="cm-property">pose</span>.<span class="cm-
property">position</span>.<span class="cm-property">y</span> = <span class="cm-number">8</span>
<span class="cm-variable">resp</span> = <span class="cm-variable">srv_client_set_model_state</span>(<span class="cm-
variable">model_state</span>)
```

We can incorporate this same code (which is actually done inside **bug1.py**) to move the robot at a desired location before starting our navigation algorithm. Finally, we can test the changes made by launching the project. Start the simulation from **Simulations > Select launch file… > my_worlds > bug1.launch**. Then execute the **bug1** algorithm by opening **Tools > Shell** and enter commands

```
$ roslaunch motion_plan bug1.launch
```

Now you will see the robot navigating towards the goal. To modify the robot initial position and goal position, we have defined arguments in this launch file (**bug1.launch**). That finishes this part.

REFERENCES:

RDS: ROS DEVELOPMENT STUDIO

SIMULATION: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT-SIMULATION

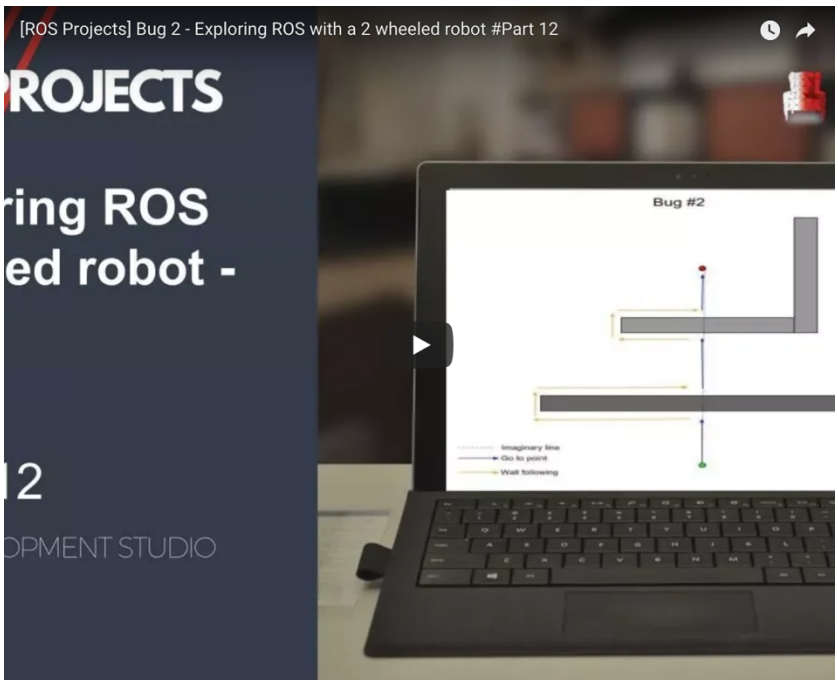
MOTION PLANNING: HTTPS://BITBUCKET.ORG/THECONSTRUCTCORE/TWO-WHEELED-ROBOT-MOTION-PLANNING

ROS COURSES: ROBOT IGNITE ACADEMY

Bug 2 - Exploring ROS with a 2 wheeled robot #Part 12

Motion planning algorithms - In this video we show the implemented code for Bug 2 behavior and a simulation using it as well. From planning the line between starting and desired points, going straight to the point and following obstacles. Using the same robot and code we have been developing in this series.

Video Tutorial Part 12



Steps to recreate the project as shown in the video

Step 12.1

Lets start by creating a new project on RDS and cloning the project files from online repository.

- Go to ROS Development Studio and create a new project.

- Provide a suitable project name and some useful description. (We have named the project **exploring_ros_video_12**)
- Load/Start the project.

Now We will fetch the project files we have developed in previous part:

- Clone the *bitbucket* repository two-wheeled-robot - Simulation.
- Checkout to the **right** branch (here is more information about branch and branching in git)
- Open **Tools > Shell** and run the following commands

```
$ cd simulation_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-
wheeled-robot-simulation.git
```

- Compile the project to make it ready to use

```
cd ~/simulation_ws/src
catkin_make
```

- Clone another github repository two-wheeled-robot - Motion Planning. Open **Tools > Shell** and run the following commands

```
$ cd catkin_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-
wheeled-robot-motion-planning.git
$ cd two-wheeled-robot-motion-planning
$ git checkout 389faca
```

- Compile the project to make it ready to use

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

Step 12.2

- Run a simulation. Open **Simulations > Select launch file... > my_worlds > world2.launch**.
- Open the script **bug2.py** located inside **~catkin_ws/src/two-wheeled-robot-motion-planning/scripts** directory

This script (**bug2.py**) contains the code for the new **Bug 2** navigation algorithm. Lets us analyze the contents of this script. Here are the contents of this file for reference

```
import rospy
# import ros message
from geometry_msgs.msg import Point
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import Odometry
from tf import transformations
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState
# import ros service
from std_srvs.srv import *

import math

srv_client_go_to_point_ = None
srv_client_wall_follower_ = None
yaw_ = 0
yaw_error_allowed_ = 5 * (math.pi / 180) # 5 degrees
position_ = Point()
initial_position_ = Point()
initial_position_.x = rospy.get_param('initial_x')
initial_position_.y = rospy.get_param('initial_y')
initial_position_.z = 0
desired_position_ = Point()
desired_position_.x = rospy.get_param('des_pos_x')
desired_position_.y = rospy.get_param('des_pos_y')
desired_position_.z = 0
regions_ = None
state_desc_ = ['Go to point', 'wall following']
state_ = 0
count_state_time_ = 0 # seconds the robot is in a state
count_loop_ = 0
# 0 - go to point
# 1 - wall following

# callbacks
def clbk_odom(msg):
    global position_, yaw_

    # position
    position_ = msg.pose.pose.position

    # yaw
    quaternion = (
```

```
        msg.pose.pose.orientation.x,
        msg.pose.pose.orientation.y,
        msg.pose.pose.orientation.z,
        msg.pose.pose.orientation.w)
    euler = transformations.euler_from_quaternion(quaternion)
    yaw_ = euler[2]

def clbk_laser(msg):
    global regions_
    regions_ = {
        'right':  min(min(msg.ranges[0:143]), 10),
        'fright': min(min(msg.ranges[144:287]), 10),
        'front':  min(min(msg.ranges[288:431]), 10),
        'fleft':  min(min(msg.ranges[432:575]), 10),
        'left':   min(min(msg.ranges[576:719]), 10),
    }

def change_state(state):
    global state_, state_desc_
    global srv_client_wall_follower_, srv_client_go_to_point_
    global count_state_time_
    count_state_time_ = 0
    state_ = state
    log = "state changed: %s" % state_desc_[state]
    rospy.loginfo(log)
    if state_ == 0:
        resp = srv_client_go_to_point_(True)
        resp = srv_client_wall_follower_(False)
    if state_ == 1:
        resp = srv_client_go_to_point_(False)
        resp = srv_client_wall_follower_(True)

def distance_to_line(p0):
    # p0 is the current position
    # p1 and p2 points define the line
    global initial_position  , desired  position
    p1 = initial_position_
    p2 = desired_position
    # here goes the equation
    up_eq = math.fabs((p2.y - p1.y) * p0.x - (p2.x - p1.x) * p0.y + (p2.x * p1.y) - (p2.y * p1.x))
    lo_eq = math.sqrt(pow(p2.y - p1.y, 2) + pow(p2.x - p1.x, 2))
    distance = up_eq / lo  eq

    return distance


def normalize_angle(angle):
    if(math.fabs(angle) > math.pi):
        angle = angle - (2 * math.pi * angle) / (math.fabs(angle))
    return angle

def main():
    global regions_, position_, desired_position_, state_, yaw_, yaw_error_allowed_
    global srv_client_go_to_point_, srv_client_wall_follower_
    global count_state_time_, count_loop_

    rospy.init_node('bug0')

    sub_laser = rospy.Subscriber('/m2wr/laser/scan', LaserScan, clbk_laser)
    sub_odom = rospy.Subscriber('/odom', Odometry, clbk_odom)

    rospy.wait_for_service('/go_to_point_switch')
```

```
106    rospy.wait_for_service('/wall_follower_switch')
107    rospy.wait_for_service('/gazebo/set_model_state')
108
109    srv_client_go_to_point_ = rospy.ServiceProxy('/go_to_point_switch', SetBool)
110    srv_client_wall_follower_ = rospy.ServiceProxy('/wall_follower_switch', SetBool)
111    srv_client_set_model_state = rospy.ServiceProxy('/gazebo/set_model_state', SetModelState)
112
113    # set robot position
114    model_state = ModelState()
115    model_state.model_name = 'm2wr'
116    model_state.pose.position.x = initial_position_.x
117    model_state.pose.position.y = initial_position_.y
118    resp = srv_client_set_model_state(model_state)
119
120    # initialize going to the point
121    change_state(0)
122
123    rate = rospy.Rate(20)
124    while not rospy.is_shutdown():
125        if regions_ == None:
126            continue
127
128        distance_position_to_line = distance_to_line(position_)
129
130        if state_ == 0:
131            if regions_['front'] > 0.15 and regions_['front'] < 1:
132                change_state(1)
133
134        elif state_ == 1:
135            if count_state_time_ > 5 and \
136               distance_position_to_line < 0.1:
137                change_state(0)
138
139        count_loop_ = count_loop_ + 1
140        if count_loop_ == 20:
141            count_state_time_ = count_state_time_ + 1
142            count_loop_ = 0
143
144        rospy.loginfo("distance to line: [%.2f], position: [%.2f, %.2f]", distance_to_line(position_), position_.x, position_.y)
145        rate.sleep()
146
147 if __name__ == "__main__":
148    main()
```

- The number of states is again 2 (as in **Bug 0**) Go to point and wall following.

- We have one more function **distance_to_line()**. It calculates the distance of the robot from the imaginary line that joins **initial position** of the robot with the **desired position** of the robot.

- The idea of **Bug 2** algorithm is to follow this imaginary line in absence of obstacle (remember go to point). When an obstacle shows up, the robot starts circumnavigating it till it again finds itself close to the imaginary line.

- Rest of the code is similar to those of the previous parts (callbacks and state transition logic)

- Also note that we have made use of the count_state_time_ variable to track time elapsed since changing state. This helps us to wrongly triggering state transition on the first contact with the imaginary line on first encounter. We let some time go by before we seek to find the imaginary line after we have started circumnavigating the obstacle.

Now we can go ahead and run the code. Open **Tools > Shell** and enter commands

```
$ roslaunch motion_plan bug2.launch
```

The robot in the simulation will start the robot motion. With that we have finished the part 12.

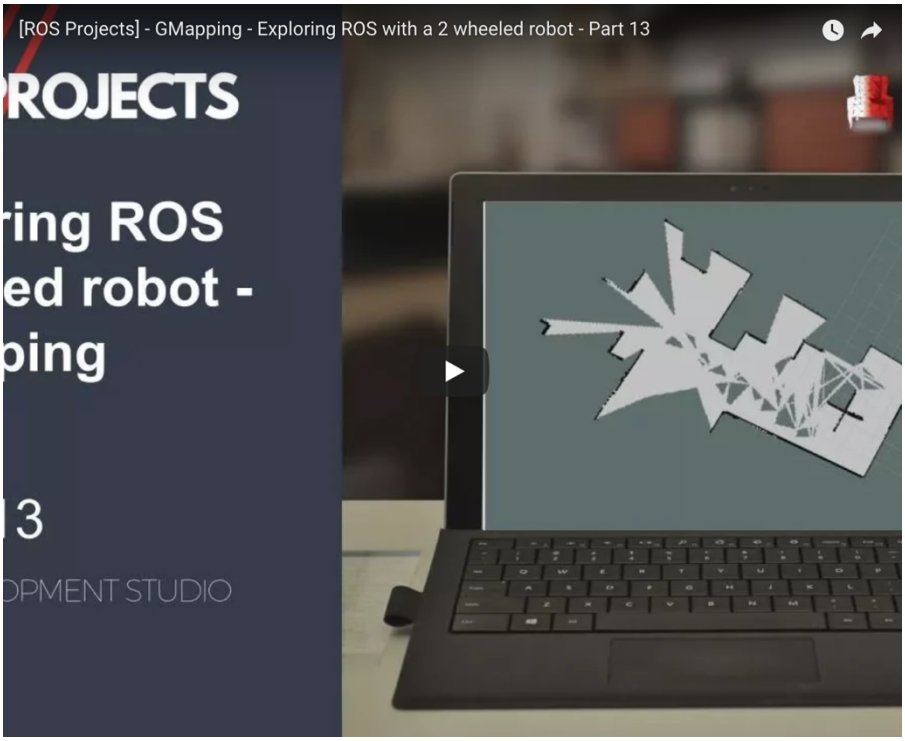
References
Robot Ignite Academy: Robot Ignite Academy

ROS Development Studio: RDS
Simulation: https://bitbucket.org/theconstructcore/two-wheeled-robot
Motion package: https://bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning

GMapping - Exploring ROS with a 2 wheeled robot #Part 13

In this video we are going to use ROS GMapping in our 2 wheeled robot, the one used in the previous videos, to generate a map using SLAM technique. We are using the robot Laser Scan and Odometry data to generate the map.

Video Tutorial Part 13



Steps to recreate the project as shown in the video

Step 13.1

Lets start by creating a new project on RDS and cloning the project files from online repository.

- Go to ROS Development Studio and create a new project.

- Provide a suitable project name and some useful description. (We have named the project **exploring_ros_video_13**)
- Load/Start the project.

Now We will fetch the project files we have developed in previous part:

- Clone the bitbucket repository two-wheeled-robot - Simulation.
- Checkout to the **right** branch (here is more information about branch and branching in git)
- Open **Tools > Shell** and run the following commands

```
$ cd simulation_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-simulation.git
```

- Compile the project to make it ready to use

```
cd ~/simulation_ws/src
catkin_make
```

- Clone another github repository two-wheeled-robot - Motion Planning. Open **Tools > Shell** and run the following commands

```
$ cd catkin_ws/src
$ git clone
https://marcoarruda@bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning.git
```

- Compile the project to make it ready to use

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

Step 13.2

In this part we are going to work with the gmapping package. The **Gmapping** package package helps in creating the map of the robot environment and it requires the following information to create environment map

- Laser scan
- Odometry information
- Sensor to robot base transform (named **link_chassis** in our robot urdf model)

The project we cloned contains the **launch file** to start the project. Open **Tools > IDE**, browse to **~catkin_ws/src/two-wheeled-robot-motion-planning/launch/** directory and load the **gmapping.launch** file. Here are the contents of this file for reference
The various elements and their utility is discussed next

Topics:

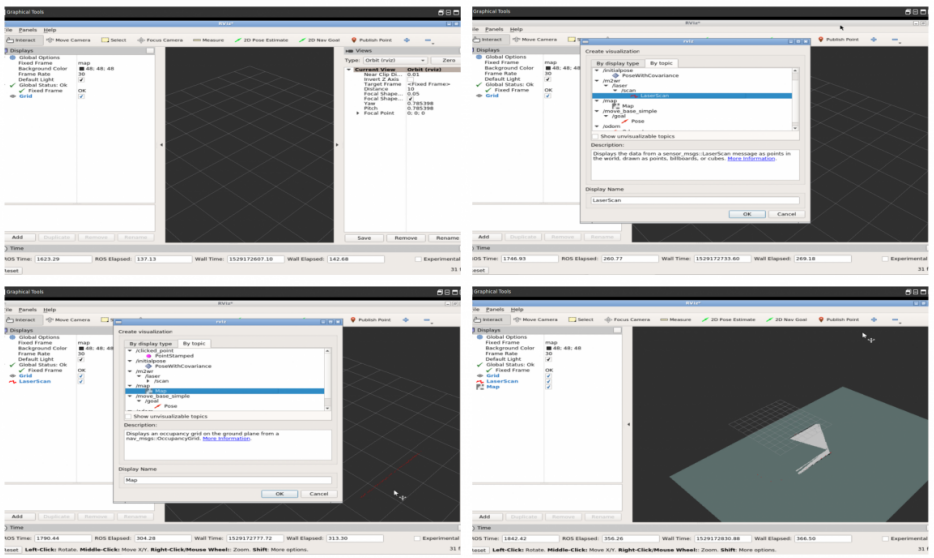
- scan_topic: points to the laser scan topic (**/m2wr/laser/scan**)
- base_frame: points to the robot base element (link_chassis)
- odom_frame: point to the odometry topic (**/odom**)

Transform:

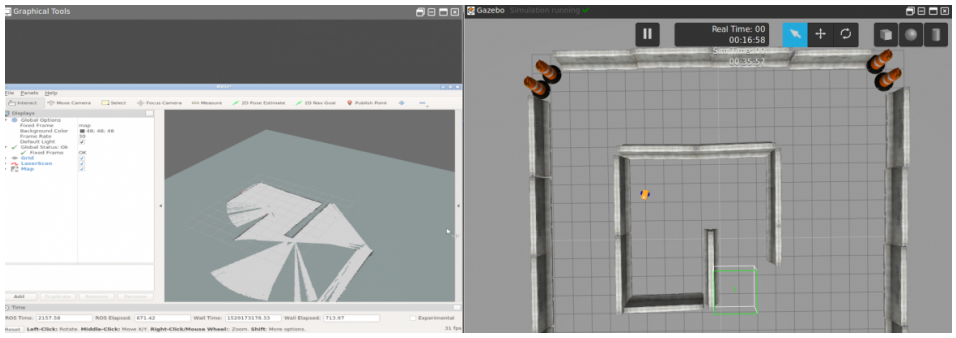
- robot_state_publisher : publishes the transform of laser scanner with respect to robots chassis (**link_chassis**)
- Rviz: To visualize the map data and robot.
- Gmapping node: This node is responsible for doing the mapping. We have specified various important arguments to use this package. Most of these values come from the example given in the wiki.

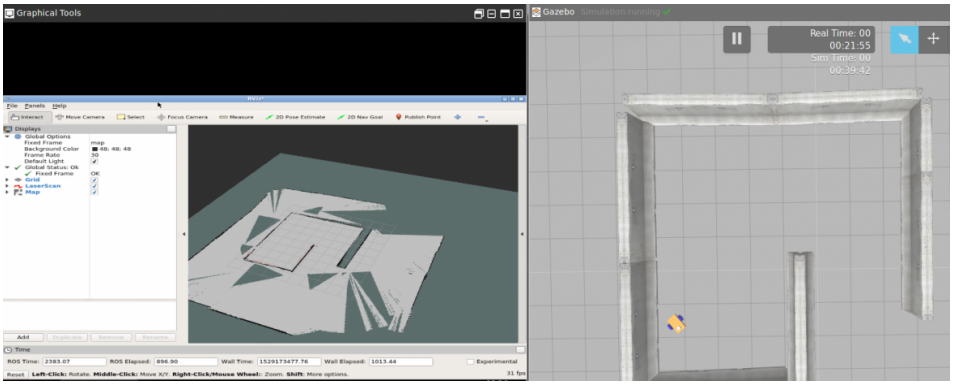
Let’s launch the simulation and see the robot in action. Load the **Bug 1** scene, open **Simulations > Select launch file... > my_worlds > bug1.launch**. We need to start a **Graphical Tool** from **Tools > Graphical Tools** menu to see the map in **rviz**. Also we will need a **Shell** to launch the **Gmapping** package.

Once the **rviz** window appears in the **Graphical Tools**, use the **Add** button (left bottom) to add a **laser_scan** and **map** display to the pane.



Now we can start another **Shell** and start the **Bug 1** algorithm to make the robot move. While the robot moves and registers the laser scan data we will see the map building.





As the robot moves around in the world we see the map coming up. With this we have finishes the series. To learn more concepts ROS use the following resources.

Robot Ignite Academy: https://goo.gl/DCR2EA
ROS Development Studio: https://goo.gl/aW7x5y

Repositories:
Simulation: https://bitbucket.org/theconstructcore/two-wheeled-robot-simulation
Motion Planning: https://bitbucket.org/theconstructcore/two-wheeled-robot-motion-planning

What Next?

You can keep learning ROS with the following courses:

- URDF
- ROS Basics
- ROS Navigation