Final Project Report

Autonomous Navigation

SPC 418: Control Systems Design for Autonomous Vehicles

University of Science and Technology, Zewail City

Aerospace Engineering

Fall 2018

Table of Contents

1. Problem Statement

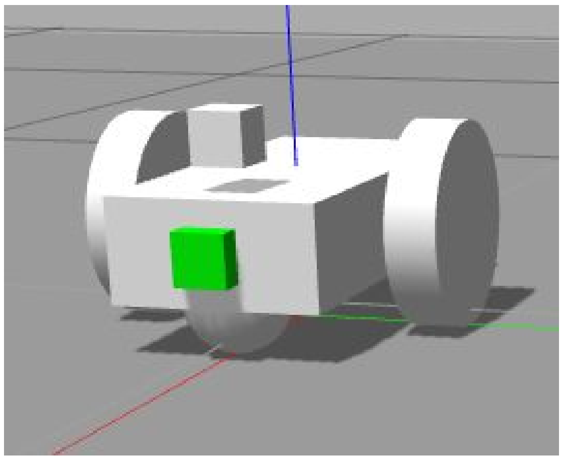
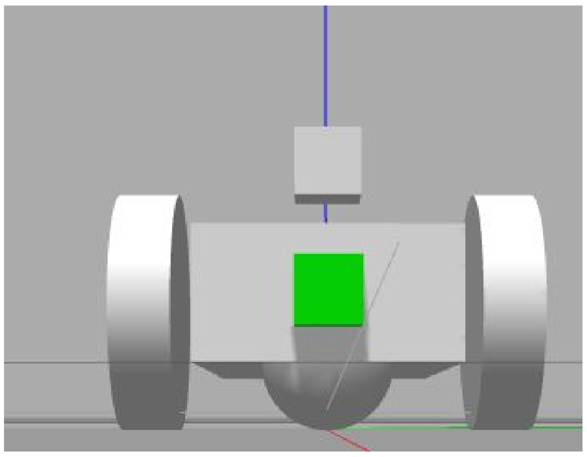
We are required to simulate a robot that is capable of autonomous navigation of a given map. Further, the robot is supposed to be able to avoid collision with statics obstacles.

2. Solution Strategy

The solution is divided into main steps that begin with deciding on the robot structure then adding the plugins. The next steps include creating the assigned map and launching the robot in this map. After that we will try to control the robot with the keyboard and we will start mapping our world. The final step will include the navigation process from any position in the map to another position without colliding with the existing objects.

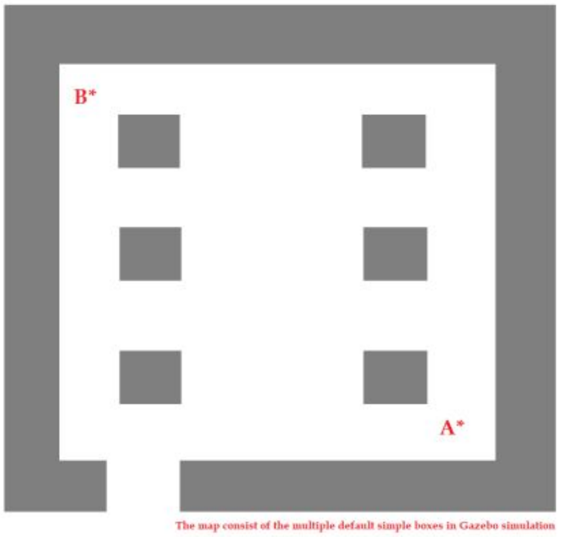
3. Robot Structure

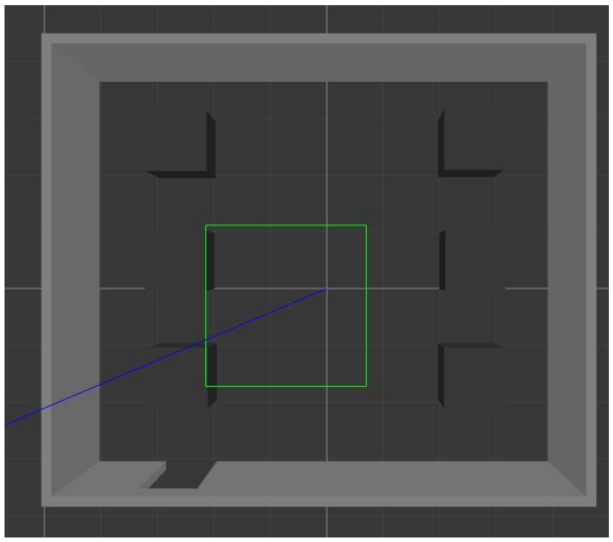
The robot is divided into three main parts. The main part is the robot body which connects all the robot parts together. The second part is the wheel, two of them are defined and both are attached to the main body by two joints, one for each wheel. The final part is the box used to symbolize the LIDAR sensor and it is attached to the body through a joint. All of these links and joints are defined in the bot.gazebo and bot.xacro files discussed in the plugin section.

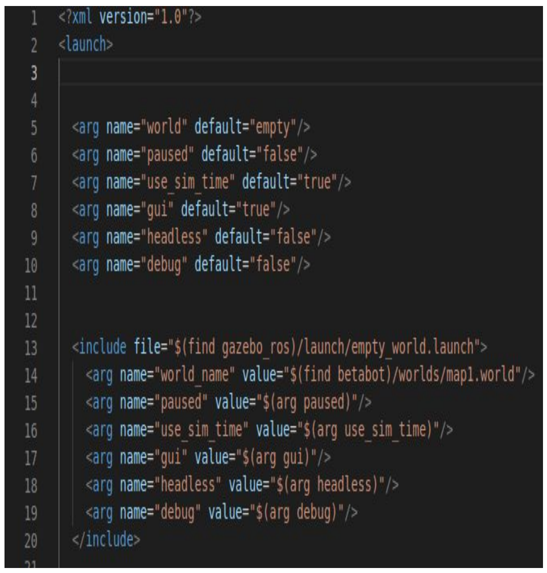
4. Environment

We were assigned map 1 for our project which is shown below. To construct a similar map a 2 step process was used; first, the boxes are dropped in the Gazebo world, and their positions are adjusted in their properties tab. The second step is to construct the walls in building editor and are then inserted in Gazebo. Then the environment is saved for later use as a ".world" file. Both the assigned map and the created map are shown below, side by side. Also the world created can be found in the worlds folder inside the package folder as shown below.

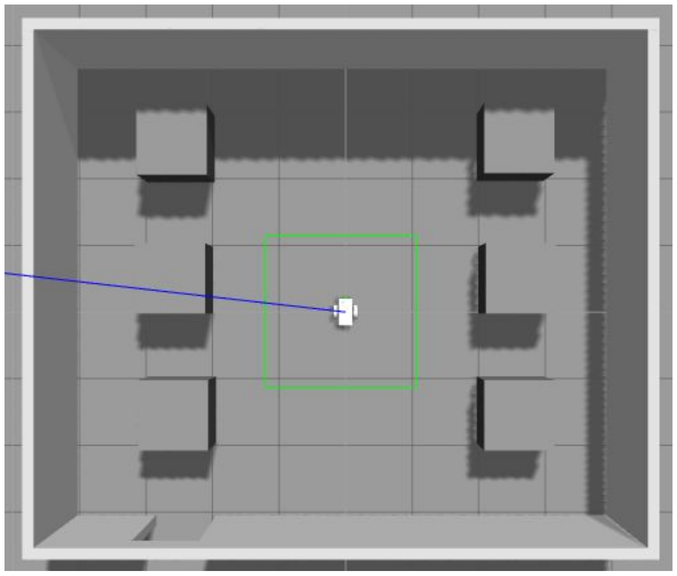

B*
A*
The map consist of the multiple default simple boxes in Gazebo simulation



The next step, before adding the plugins, is to launch the robot in the created world in gazebo and to make sure that no errors occur in the process, which is shown in the following picture.



```
<?xml version="1.0"?>
<launch>


  <arg name="world" default="empty"/>
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>


  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find betabot)/worlds/map1.world"/>
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="gui" value="$(arg gui)"/>
    <arg name="headless" value="$(arg headless)"/>
    <arg name="debug" value="$(arg debug)"/>
  </include>
```

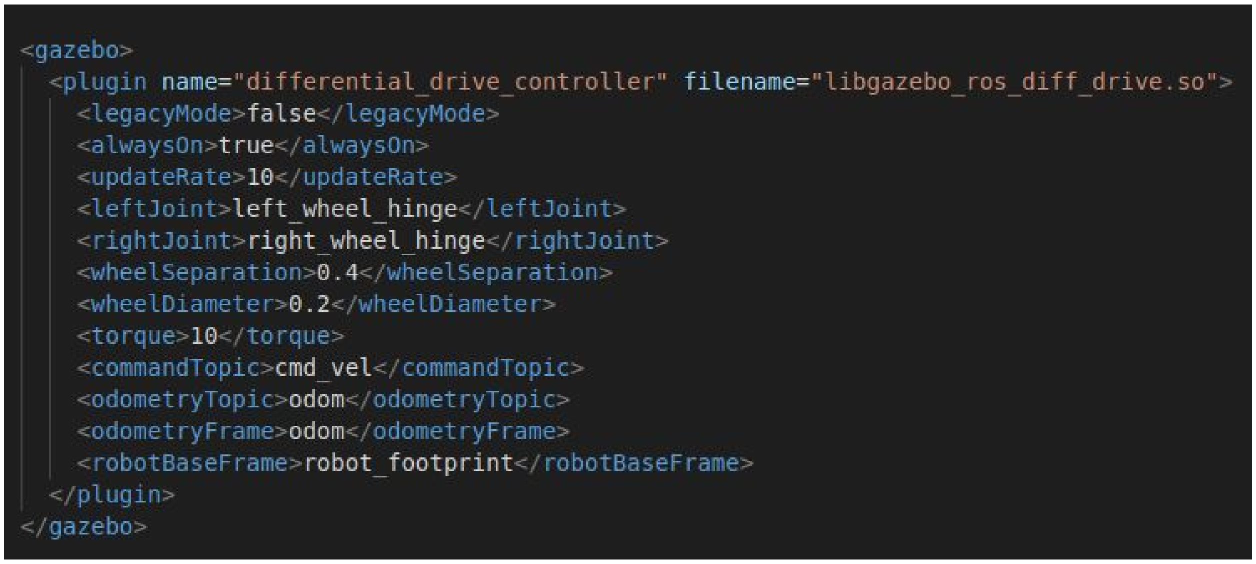


5. Plugins and Sensors

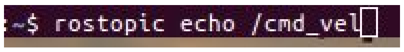
Four plugins were added to our robot: a differential drive plugin, an IMU, a camera, and a laser scanner. These were added using Gazebo tutorials and are discussed in the following subsections.

5.1 Differential Drive

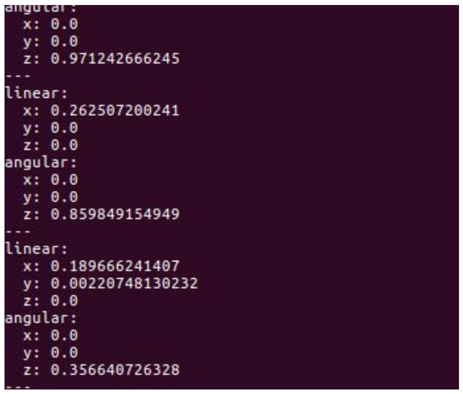
The differential drives subscribes to the commands sent to "cmd_vel" topic and publishes on "odom" topic. The code below is put in bot.gazebo file inside the URDF folder, and the parameters are adjusted to match those of the robot used. To make sure that it works, type the command shown below and notice the velocity reading in the terminal. You can also make sure that it works via a twist message, which will be shown below in the teleop section.



```
<gazebo>
  <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
    <legacyMode>false</legacyMode>
    <alwaysOn>true</alwaysOn>
    <updateRate>10</updateRate>
    <leftJoint>left_wheel_hinge</leftJoint>
    <rightJoint>right_wheel_hinge</rightJoint>
    <wheelSeparation>0.4</wheelSeparation>
    <wheelDiameter>0.2</wheelDiameter>
    <torque>10</torque>
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <robotBaseFrame>robot_footprint</robotBaseFrame>
  </plugin>
</gazebo>
```



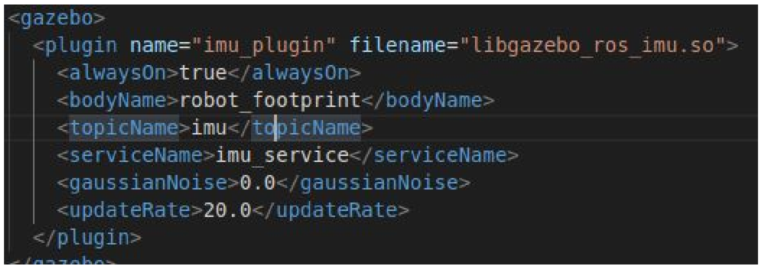
```
:~$ rostopic echo /cmd_vel
```



```
angular:
  x: 0.0
  y: 0.0
  z: 0.971242666245
---
linear:
  x: 0.262507200241
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.859849154949
---
linear:
  x: 0.189666241407
  y: 0.00220748130232
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.356640726328
---
```

5.2 IMU

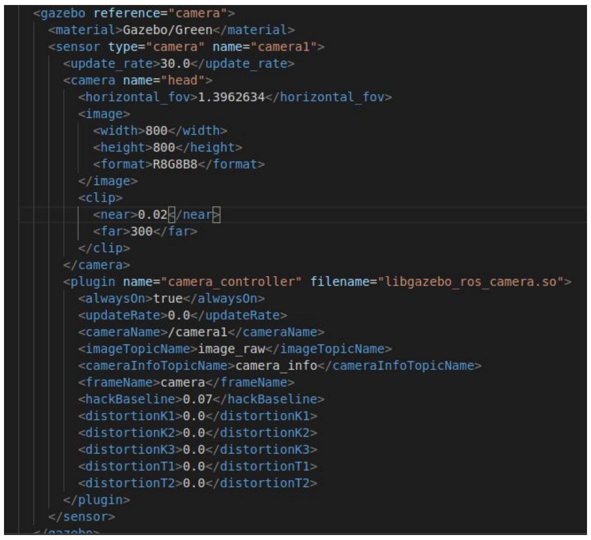
The IMU is added as the differential drive. It publishes its value on “/imu”. The plugin code is shown below.



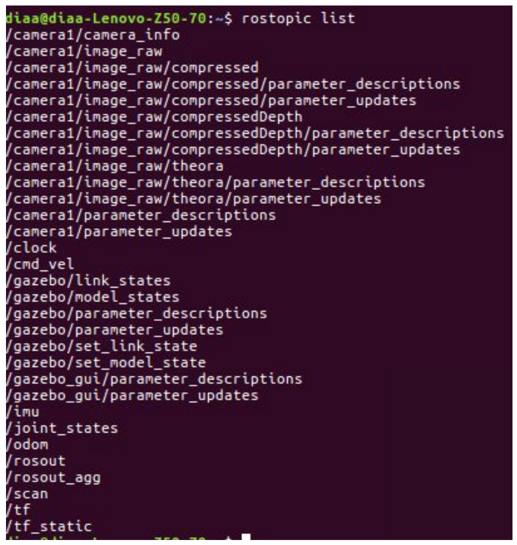
```
<gazebo>
  <plugin name="imu_plugin" filename="libgazebo_ros_imu.so">
    <alwaysOn>true</alwaysOn>
    <bodyName>robot_footprint</bodyName>
    <topicName>imu</topicName>
    <serviceName>imu_service</serviceName>
    <gaussianNoise>0.0</gaussianNoise>
    <updateRate>20.0</updateRate>
  </plugin>
```

5.3 Camera

The camera plugin is added exactly as the differential drive. To make sure that it and the other plugins, are recognized we use the command rostopic list. The plugin code and the topic list are shown below.



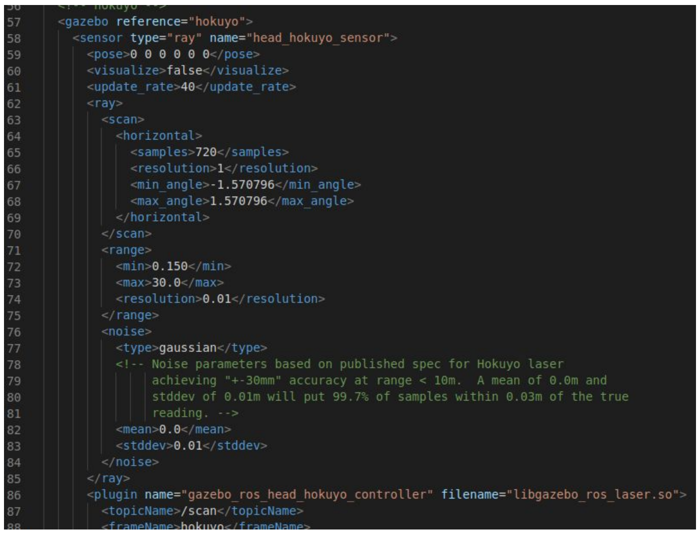
```
<gazebo reference="camera">
  <material>Gazebo/Green</material>
  <sensor type="camera" name="camera1">
    <update_rate>30.0</update_rate>
    <camera name="head">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>800</width>
        <height>800</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>/camera1</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>camera</frameName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
```



```
diaa@diaa-Lenovo-Z50-70:~$ rostopic list
/camera1/camera_info
/camera1/image_raw
/camera1/image_raw/compressed
/camera1/image_raw/compressed/parameter_descriptions
/camera1/image_raw/compressed/parameter_updates
/camera1/image_raw/compressedDepth
/camera1/image_raw/compressedDepth/parameter_descriptions
/camera1/image_raw/compressedDepth/parameter_updates
/camera1/image_raw/theora
/camera1/image_raw/theora/parameter_descriptions
/camera1/image_raw/theora/parameter_updates
/camera1/parameter_descriptions
/camera1/parameter_updates
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_gui/parameter_descriptions
/gazebo_gui/parameter_updates
/imu
/joint_states
/odom
/rosout
/rosout_agg
/scan
/tf
/tf_static
```

5.4 Laser Scanner

The laser scanner publishes its readings on “scan” topic. We are using “ray” type not “gpu_ray” because our Nvidia drivers are not working correctly. The field of view angles are set to [-90,90] degrees, and the range is set to 30 m with resolution of 0.01 m.

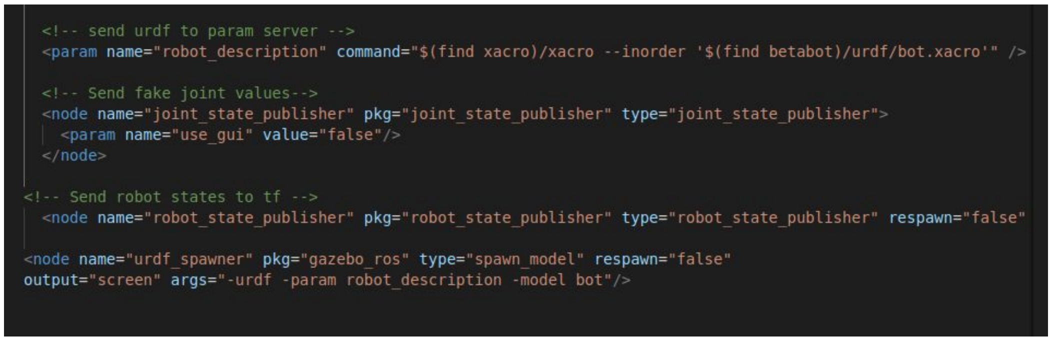


```
  <!-- hokuyo -->
  <gazebo reference="hokuyo">
    <sensor type="ray" name="head_hokuyo_sensor">
      <pose>0 0 0 0 0 0</pose>
      <visualize>false</visualize>
      <update_rate>40</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>720</samples>
            <resolution>1</resolution>
            <min_angle>-1.570796</min_angle>
            <max_angle>1.570796</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.150</min>
          <max>30.0</max>
          <resolution>0.01</resolution>
        </range>
        <noise>
          <type>gaussian</type>
          <!-- Noise parameters based on published spec for Hokuyo laser
               achieving "+-30mm" accuracy at range < 10m.  A mean of 0.0m and
               stddev of 0.01m will put 99.7% of samples within 0.03m of the true
               reading. -->
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
        <topicName>/scan</topicName>
        <frameName>hokuyo</frameName>
```

6. RVIZ Launch

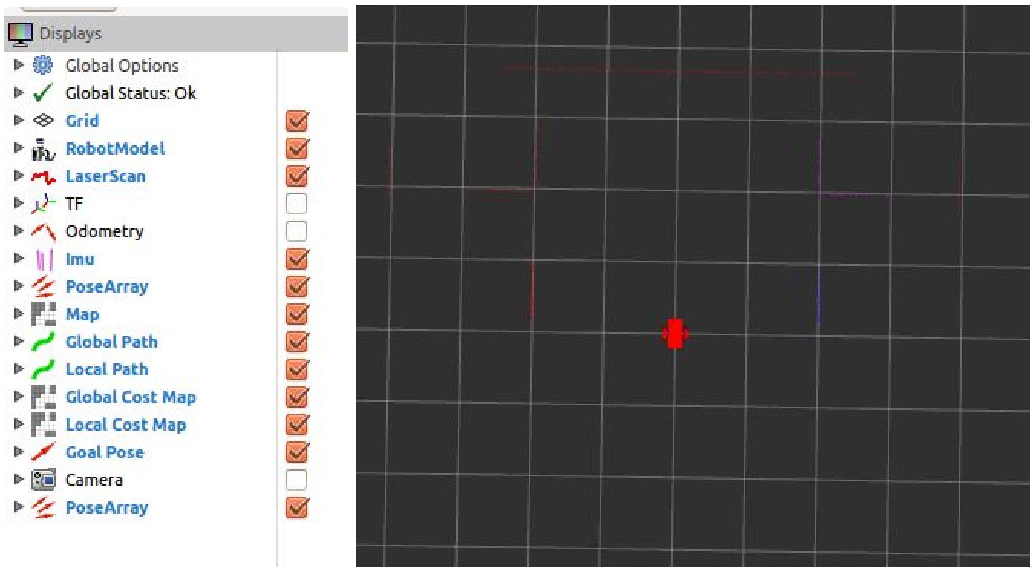
Up to this moment we have 2 files in the URDF folder representing the robot and its plugins. We also have a world representing map1 in the worlds folder and finally a gazebo launch file in the launch folder to launch gazebo. The next step is to make sure that rviz reads all the connections in a correct manner.

In order to be able to publish the state of the robot and its joints, we add "robot state publisher" and "joint state publisher" nodes to the launch file. Those nodes use "robot description parameter" which should also be defined.



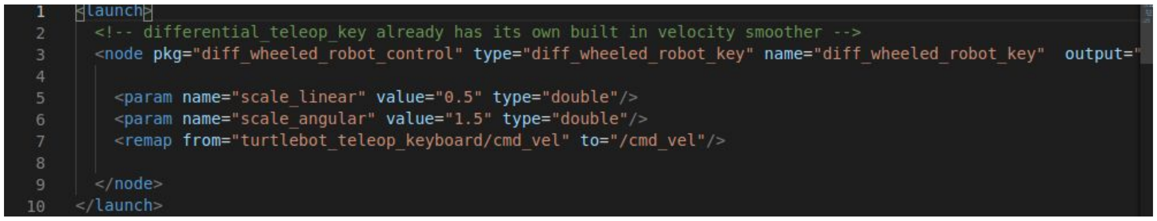
```
  <!-- send urdf to param server -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find betabot)/urdf/bot.xacro'" />

  <!-- Send fake joint values-->
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
    <param name="use_gui" value="false"/>
  </node>

<!-- Send robot states to tf -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false"

<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false"
output="screen" args="-urdf -param robot_description -model bot"/>
```

Then Rviz is run using `$ rosrun rviz rviz`. The below pictures shows the robot launched in RVIZ with the LIDAR and the other sensors added to it. Other added materials are shown in the below picture which include the TF, the map, the camera, and others discussed in later sections.


Displays
- Global Options
- Global Status: Ok
- Grid ✓
- RobotModel ✓
- LaserScan ✓
- TF ☐
- Odometry ☐
- Imu ✓
- PoseArray ✓
- Map ✓
- Global Path ✓
- Local Path ✓
- Global Cost Map ✓
- Local Cost Map ✓
- Goal Pose ✓
- Camera ☐
- PoseArray ✓

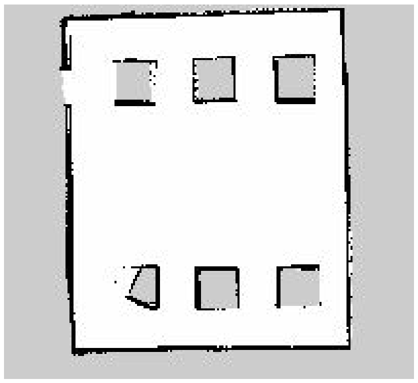
7. Teleop Keyboard Control

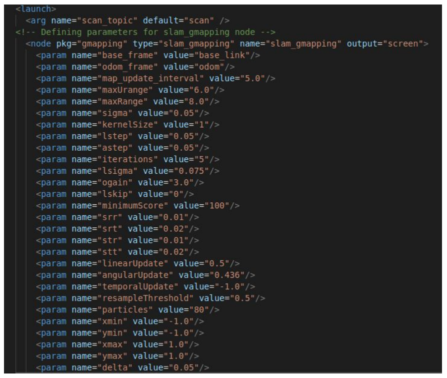
We need to control the robot to be able to navigate the environment and create the map that the robot will use in autonomous navigation. There are two ways to do that. The first ways includes giving orders directly to the robot to move to certain known points using the commands such as rostopic pub /cmd_vel geometry_msgs/Twist [2.0,0.0,0.0] [5.0,1.0,0.0]. The second includes creating a teleop launch file that helps in controlling the robot using the keyboard. Reference [1] was used to install the dependencies and then the launch file was created as shown below.


```
<launch>
  <!-- differential_teleop_key already has its own built in velocity smoother -->
  <node pkg="diff_wheeled_robot_control" type="diff_wheeled_robot_key" name="diff_wheeled_robot_key"  output="

    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="/cmd_vel"/>

  </node>
</launch>
```

7. G-mapping

The gmapping package is an implementation of an algorithm called Fast SLAM, which takes the laser scan data and odometry to build a 2D occupancy grid map.The slam_gmapping node subscribes the laser data and the TF data, and publishes the occupancy grid map data as output.The main parameters we need to configure are the global and local costmap parameters, the local planner, and the move_base parameters. This is done using three main steps. The first step is to launch the robot in RVIZ, and then launch the teleop file in a second terminal and the gmapping launch file in a third terminal. The second step is to manually manipulate the robot into scanning the whole map. Finally the third step is to save the scanned map using rosrun map_server map_saver -f map1 command. The next two pictures show the gmapping launch file and the map created. It can be noticed that the lower left box is not displayed correctly, however this did not create any problems in navigation in latter steps.

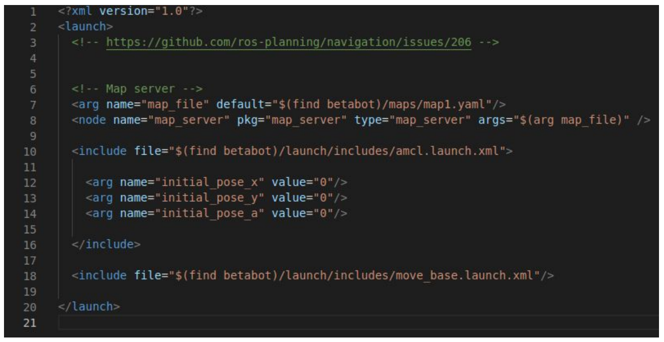




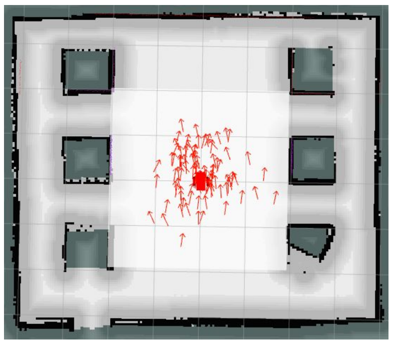
```
<launch>
  <arg name="scan_topic" default="scan" />
<!-- Defining parameters for slam_gmapping node -->
  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen">
    <param name="base_frame" value="base_link"/>
    <param name="odom_frame" value="odom"/>
    <param name="map_update_interval" value="5.0"/>
    <param name="maxUrange" value="6.0"/>
    <param name="maxRange" value="8.0"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="lskip" value="0"/>
    <param name="minimumScore" value="100"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>
    <param name="linearUpdate" value="0.5"/>
    <param name="angularUpdate" value="0.436"/>
    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="particles" value="80"/>
    <param name="xmin" value="-1.0"/>
    <param name="ymin" value="-1.0"/>
    <param name="xmax" value="1.0"/>
    <param name="ymax" value="1.0"/>
    <param name="delta" value="0.05"/>
```

8. Navigation

AMCL is a method to localize the robot in a map. It uses probability to track the position of the robot with respect to the map via a particle filter. The robot should publish a proper odometry value, TF information, and sensor data from the laser, and have a base controller and map of the surroundings. Now that the robot mapped the environment, the only thing that is needed is to add the AMCL launch file which will help localize the robot position. The next step is to send the goal position to the move base which will send the goal to a goal planner that will decide on a collision free path. This path is with respect to a global map planner, the global map planner then sends the data to a local map planner to execute each segment separately. A picture of the launch file is shown below, and another that shows the robot launched in RVIZ with the amcl file launched.

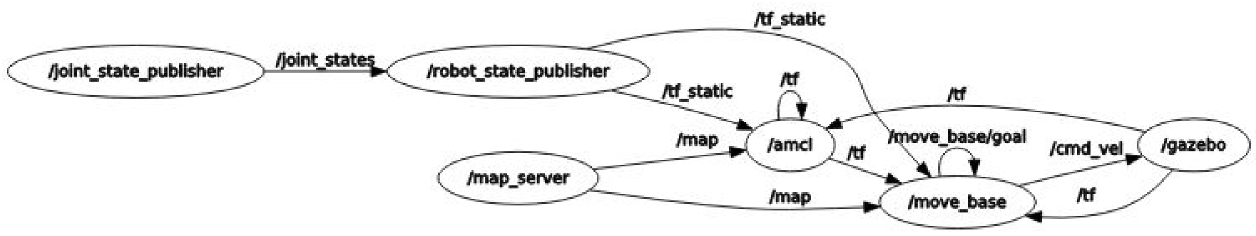


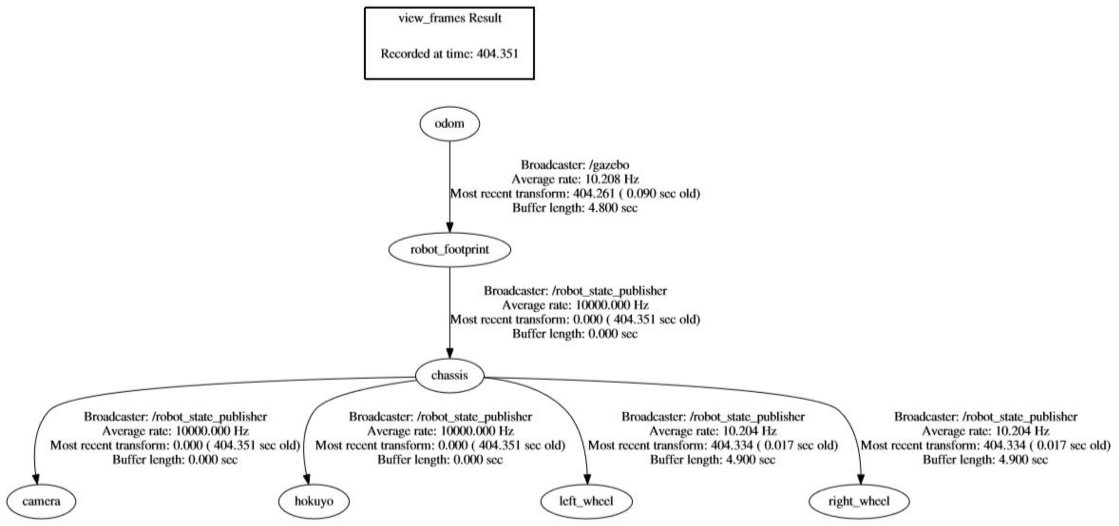
```
<?xml version="1.0"?>
<launch>
  <!-- https://github.com/ros-planning/navigation/issues/206 -->


  <!-- Map server -->
  <arg name="map_file" default="$(find betabot)/maps/map1.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

  <include file="$(find betabot)/launch/includes/amcl.launch.xml">

    <arg name="initial_pose_x" value="0"/>
    <arg name="initial_pose_y" value="0"/>
    <arg name="initial_pose_a" value="0"/>

  </include>

  <include file="$(find betabot)/launch/includes/move_base.launch.xml"/>

</launch>
```



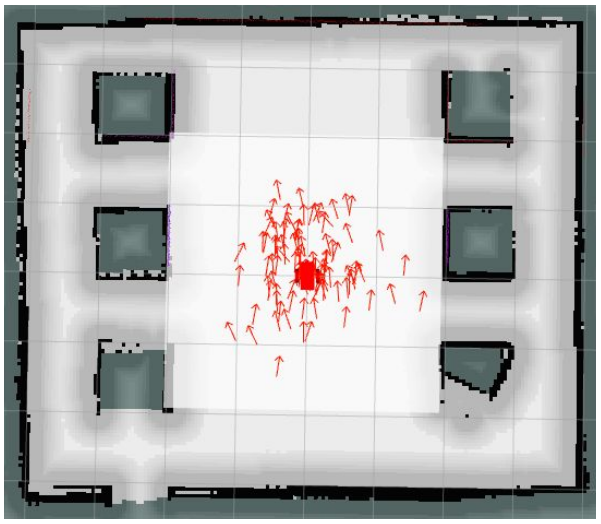
9. Results

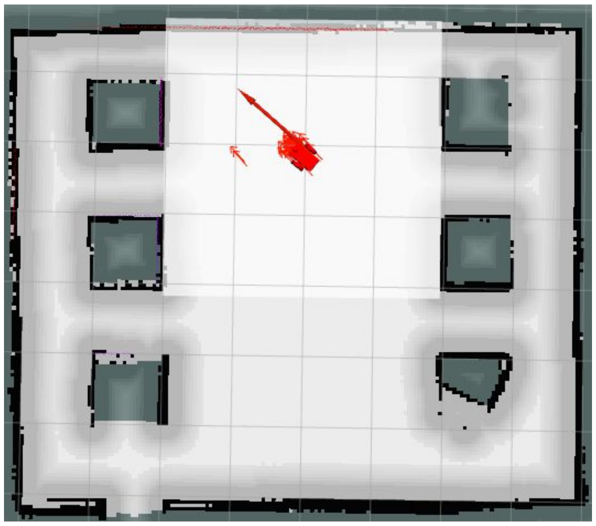
A couple of important charts are shown below. They show the connections between links and joints of the robot and the connection between all the publishers and subscribers of the complete project.

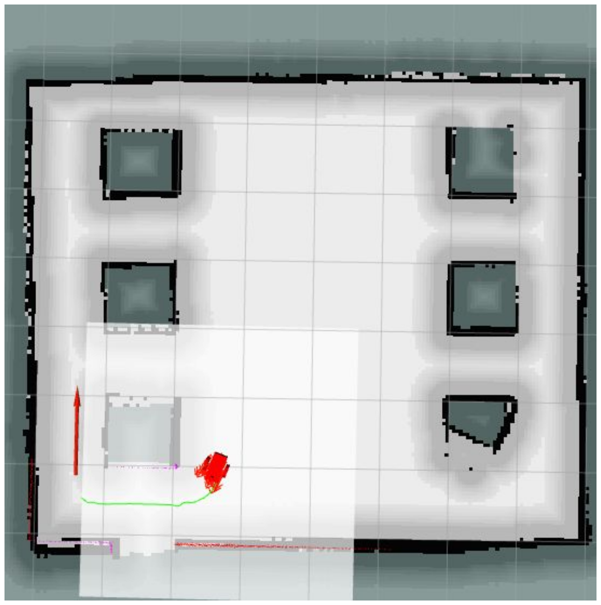



view_frames Result
Recorded at time: 404.351
odom
Broadcaster: /gazebo
Average rate: 10.208 Hz
Most recent transform: 404.261 (0.090 sec old)
Buffer length: 4.800 sec
robot_footprint
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (404.351 sec old)
Buffer length: 0.000 sec
chassis
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (404.351 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (404.351 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 404.334 (0.017 sec old)
Buffer length: 4.900 sec
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 404.334 (0.017 sec old)
Buffer length: 4.900 sec
camera
hokuyo
left_wheel
right_wheel

At this point the robot has all what is needed to navigate autonomously. To do that launch both gazebo and amcl launch files, then run RVIZ and select 2D point NAV and place a point where you want the robot to go. Some examples are shown below.







A couples of thing that need to be noted are that the robot navigates his way around the obstacles when needed and that when given a point inside a box it gets near the box and either keeps pushing it or keeps rotating about itself. Finally, the broken box from the gmapping map is treated as a normal box and the robot behaves similarly when in its proximity.

As for the next steps that can be added to this project, one might consider taking the next step of adding another robot in the environment and adding a no collision system between the two robots to allow autonomous navigation in a dynamic environment. Another addition might be the integration of a more complex map and a robot that has more sensors for better situational awareness of the map.

10. References

[1] Cacace, J., & Joseph, L. Mastering ROS for Robotics Programming, Second Edition.