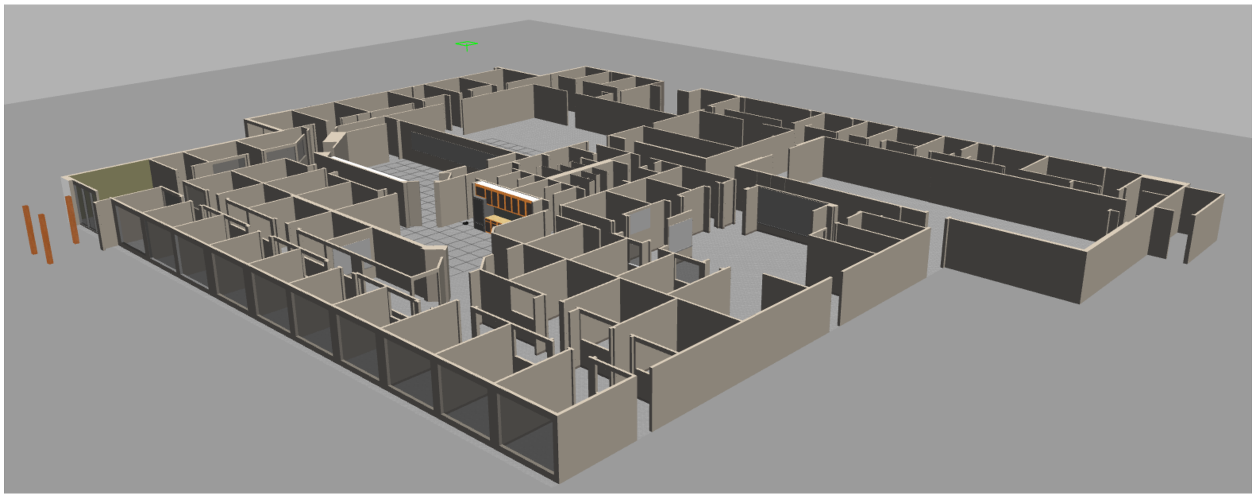
PROJECT 5: SLAM

BLACK MESA RESEARCH (Group name)

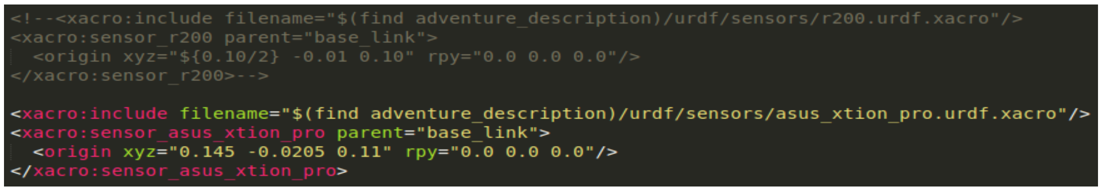
Miguel Angel Maestre Trueba and Yash Manian – CMSC498F

The objective of the project is to design an algorithm for a robot to be able to localize itself while mapping the area. In this report, all the accomplished tasks are described.

1. **Map Replacement:** The first step is to replace the Eurobot map in Gazebo with the Willow Garage map. The Willow Garage map is already in the Gazebo worlds collection. The easiest way to replace it in our package is to copy the file ***willowgarage.world*** and add it to the adventure_gazebo package and inside of this one, into the ***worlds*** folder. If we run Gazebo now, we will find that the map has changed to something like this:



2. **Robot Model:** The next step is to fix the robot model shown in Gazebo and described in the URDF files. For this, we have to edit and change some things in ***finn.urdf.xacro*** and in ***asus_xtion_pro.urdf.xacro.*** The first is to replace the R200 camera with the Asus Xtion Pro, which is the one that the adventure turtlebots have. This is done by substituting the names as follows:



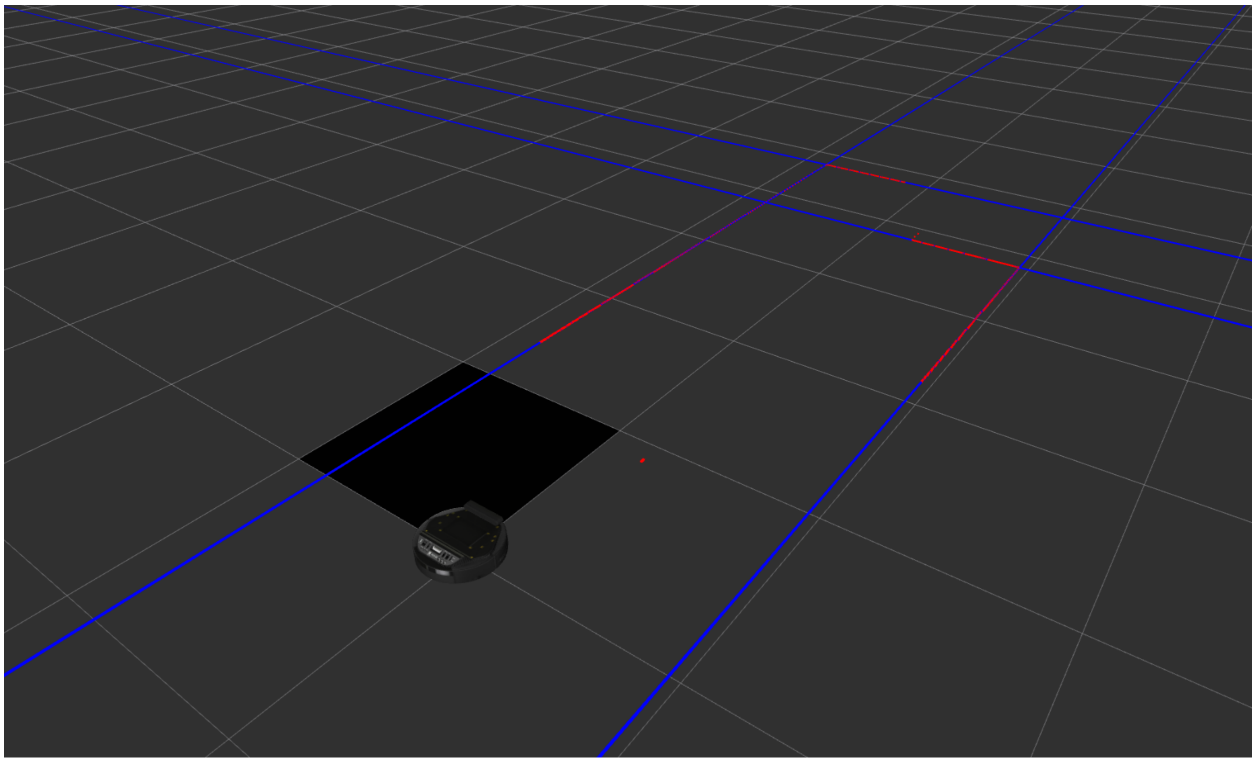
```
<!--<xacro:include filename="$(find adventure_description)/urdf/sensors/r200.urdf.xacro"/>
<xacro:sensor_r200 parent="base_link">
  <origin xyz="${0.10/2} -0.01 0.10" rpy="0.0 0.0 0.0"/>
</xacro:sensor_r200>-->

<xacro:include filename="$(find adventure_description)/urdf/sensors/asus_xtion_pro.urdf.xacro"/>
<xacro:sensor_asus_xtion_pro parent="base_link">
  <origin xyz="0.145 -0.0205 0.11" rpy="0.0 0.0 0.0"/>
</xacro:sensor_asus_xtion_pro>
```

If we launch Gazebo now, the previous camera will be replaced with the right one. But we won't be able to move it to the right location (which was measured in the real robot). By editing the Asus Xtion URDF file, we will be able to place the camera where we need to. The only thing that has to be added is in line 11 of the file, in the creation of the macro. The origin of the camera has to be read as an argument given by the Finn URDF.

Lastly, to finish the Robot Model section, ***mobile_base.launch.xml*** has to be filled in with the calls to the Kobuki and Asus drivers. The real robot (Finn) was tested by using ***roslaunch adventure_bringup minimal.launch*** and it started successfully.

3. **Launch files and parameters:** We have several thresholds and values that we want to keep track of. For that, the best thing to do is to create a *yaml* file that contains all the parameters. The launch file in the adventure_slam package contains only two commands. The first one is the one that calls the adventure_slam code and the second one is the command that loads the parameters file, so that we can access them from the code. The next step is to call this launch file in the two main launch files: *adventure_demo.launch* and *adventure.launch.* This is quite easy, we only have to add the next line:

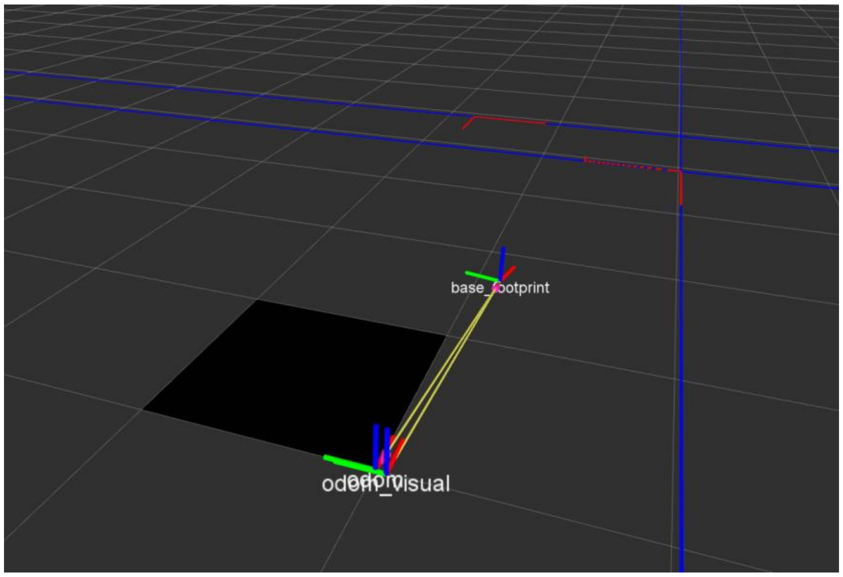
<include file="$(find adventure_slam)/launch/adventure_slam.launch"/>

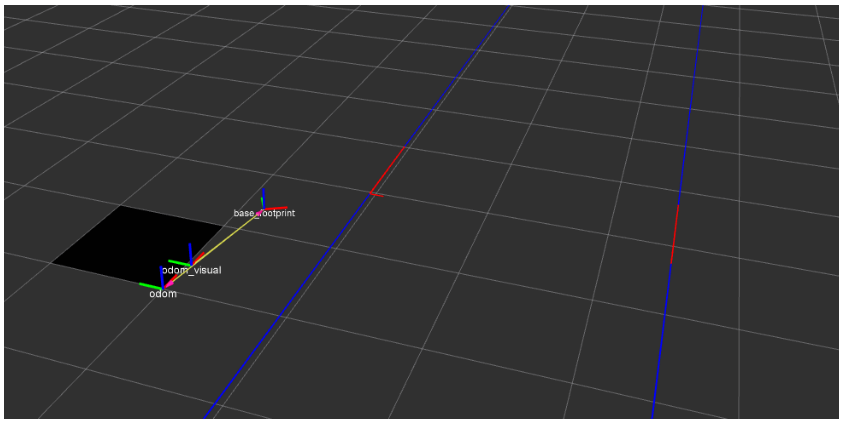


4. **Line Fitting:** For this section of the project, the Point Cloud Library (PCL) is used to fit lines for the points given by the Laser Scan. To do this, we use RANSAC. Fitting the line with the biggest amount of inliers is straightforward. To fit the rest of lines, we have to consider the points that RANSAC determines as outliers. Once we have the list of outliers, we apply RANSAC again to fit another line. This sequence is done until the number of outliers is less than a chosen threshold. In the image above, we can see the laser scans in red and their respective fitted lines in blue. The results are successful and the lines are well fitted.
5. **Localization:** Once the lines are fitted using RANSAC, we estimate the rotation and translation of the robot. We consider the fitted lines in the current and the previous frame and match them. The two conditions used are: the angle of the lines has to be under a certain value and the distance between matching lines has to be small. The first condition is only applying the angle between lines equation. It gave good results in the tests and didn't give too many problems. The second condition is more complex. To get a precise distance between matches, we create a virtual line that passes through the robot frame and is orthogonal to the lines. The intersection of this line with previous and current lines gives an approximate distance. If this distance is small enough and the angle is also small, then we consider the pair of lines to match. The matching experiments worked very well by following this conditions. The only problem comes when there are different parallel lines that are very close to each other. The matching failed sometimes in this situation.

   To get the rotation of the robot, we get the angles from the previous condition for line matching and do the average of all of them. The shift is more complex. To locate, we need an intersection between lines. So we look for intersections between line matches. This is done by using the line equation $y = mx + b$. Once the intersection is determined in previous and current frame, we get the difference between them. But since they are in different frames, we have to rotate the current intersection point to the previous frame. And once we have the distance between intersections, another rotation is applied to take it to the global frame. The average of this distances is the estimated shift of the robot.

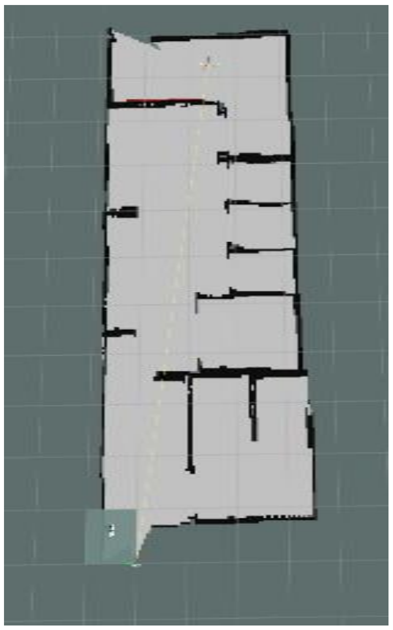
If there are no intersections between lines but there are still lines, localization won't work. But the global frame will move along the direction of the lines. To accomplish this, a virtual orthogonal line is created (as the one in the matching section). The intersections between lines and these virtual lines are used to estimate an approximate shift.
Lastly, the global shift and angle are published as transform and odometry messages.
The results of localization are quite good when there are intersections. It drifts, but not too much. When there are only non-intersecting lines, the visual_odometry frame can get far from where it is supposed to be. Also, it is important to say that localization doesn't work well when the robot's speed is high. For the tests, linear and angular speeds are always under 0.25. The following images show the results. *Base_footprint* is the robot's position, *odom_visual* is the estimated origin frame and *odom* is the wheels odometry frame, used to compare *odom_visual* with it.

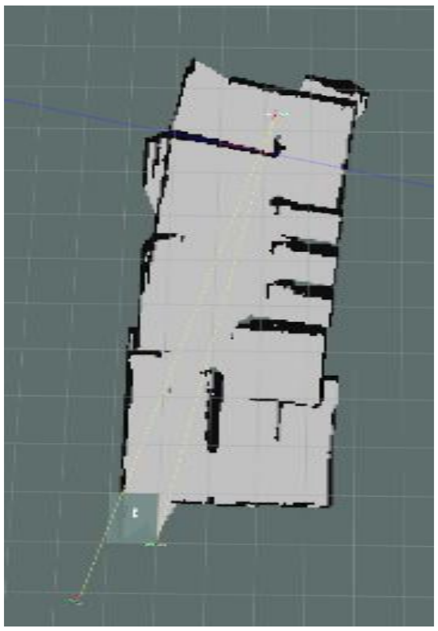


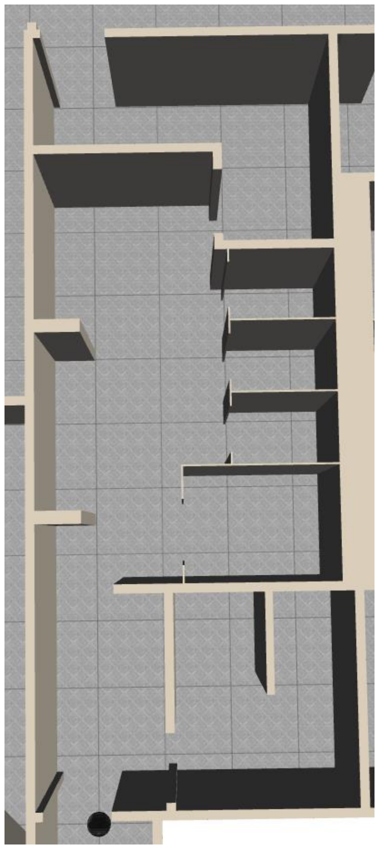


The first image shows an example of localization with intersections. The second image shows what happens to the frame when there are no intersections.

6. **Mapping:** This part is done almost exactly as in Project 4. All the codes we used are the sames ones except for ***ros_map.py***. The only difference is that we are listening to transforms instead of reading odometry messages. We also subscribe to the Laser Scan data and publish OccupancyGrid messages to create the 2D map. The images below show a mapped section of the Willow Garage world. The image to the left is using ***odom*** to create the map. The one in the middle uses ***odom_visual*** and the third one is the section of the world represented in Gazebo.







It can be seen that the mapping with ***odom*** is good but the mapping using ***odom_visual*** is not as good, mainly because of the times in which there are no line intersections. That causes the frame to drift and make the map less accurate.

7. **Conclusion:** All the parts of the project have been successfully accomplished, except for applying SLAM in the real robot to build the map of AV Williams's 4th floor. We made the

robot run the adventure_bringup package, but every time we tried to run the SLAM, it didn’t work properly.