Home Service Project

Douglas Teeple

Abstract—A Home Service Robot project implementing navigating a robot in custom-developed world in the ROS (Robot Operating System) environment is described and the results are discussed.

Index Terms—Robot, IEEEtran, Mobile Robotics, Localization, Mapping, SLAM, gazebo.

1 Introduction

The Home Service Robot project develops an environment and a mobile robot which is programmed to pick up an object at one location and deliver it to another location. The project encompasses creating a world in gazebo, mapping it using a wall following node, moving the mobile to goal locations via a path, localization and using markers to simulate picking up and moving the object.

2 Summary of Tasks

The tasks required to complete the project are:

- Design a simple environment using the Building Editor in Gazebo.
- Teleoperate the robot and manually test SLAM.
- Create a wall_follower node that autonomously drives the robot to map the environment.
- Use the 2D Nav Goal arrow in rviz to move to two different desired positions and orientations.
- Write a pick_objects node in C++ that commands the robot to move to the desired pickup and drop off zones.
- Write an add_markers node that subscribes to robot odometry keeping track of the robot pose, and publishes markers to rviz.
- Combine all of the forgoing to simulate a robot moving to a pick up point, and carrying a virtual object to a dropoff point.

3 Environment

The Building Editor tool in Gazebo was used to create a simple world customized with color. The world SDF file was saved in the catkin_ws/src/home_service_project/worlds folder.

Project Note: The directory structure suggested in the project is somewhat unusual for a ROS project. I took the liberty of modifying it as follows:

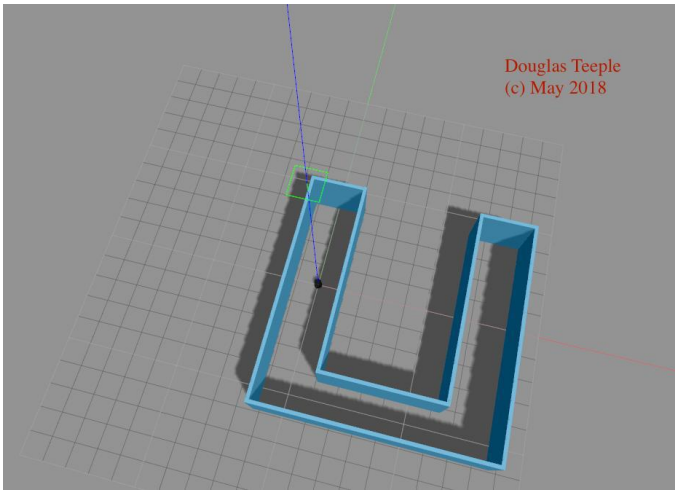

Figure 1. U world as developed in gazebo World Editor.

```
.
 add_markers.sh
 colors.sh
 doc
 home_service.sh
 images
...
 kill.sh
 launch.sh
 launch.txt
 pick_objects.sh
 src
  add_markers
   CMakeLists.txt
   include
    add_markers
   package.xml
   src
       add_markers.cpp
  depthimage_to_laserscan
...
  home_service_project
   CMakeLists.txt
   config
    amcl.yaml
    base_local_planner_params.yaml
    costmap_common_params.yaml
    global_costmap_params.yaml
    local_costmap_params.yaml
    RVIZ.rviz
   home_service_project
    CMakeLists.txt
    package.xml
   include
    home_service_project
   launch
    amcl.launch
    config
     robot_slam.rviz
    gmapping.launch
    home_service_project_world.launch
    includes
     create.launch.xml
     kobuki.launch.xml
     roomba.launch.xml
    mapping.launch
    robot_description.launch
    teleop.launch
   maps
    U.pgm
    U.yaml
   meshes
    CoffeeCup.dae
   package.xml
   scripts
    home_service_project.sh
    install_turtlebot.sh
    teleop
   src
```

```
...
  pick_objects
   CMakeLists.txt
   include
    pick_objects
   package.xml
   src
       pick_objects.cpp
  slam_gmapping
   gmapping
    CHANGELOG.rst
    CMakeLists.txt
    launch
      ...
  turtlebot_simulator
   turtlebot_gazebo
...
  turtlebot_viz
...
  wall_follower
      CMakeLists.txt
      include
       wall_follower
      package.xml
      src
       wall_follower.cpp
      wall_follower
       CMakeLists.txt
       include
        wall_follower
       package.xml
       src
test_navigation.sh
 test_slam.sh
 wall_follower.sh
```

3.1 World Creation

The world model *"U"* was created using the gazebo model building facilities by these steps:

1) Open Gazebo
2) Go to Insert -> Model Database: Select the wall template and draw the U world. Resize the U world model to be 10 meters on exterior edge and color the walls using the color palette.
3) Export the model: File -> Save World As, and saved the file as *U* in the worlds directory as described above.

4 Robot Model

The Robot was the Willow Garage http://www.willowgarage.com turlebot definition https://github.com/turtlebot/turtlebot.git and turtlebot_simulation definition https://github.com/turtlebot/turtlebot_simulator. The Kinect RGBD camera was added to the robot definition. I used the depthimage_to_laserscan node to convert to laser scans. The model has corresponding URDF and Gazebo files in the package directory.

5 Mapping

5.1 Testing SLAM

In this step I manually tested SLAM with a robot inside the U environment. I created a bash script called *test_slam.sh* to launch the SLAM test. The script automates the following steps:

Listing 1. Terminal output from test_slam.sh

```
nvidia@tegra-ubuntu:~/catkin_ws$ ./test_slam.sh
Testing SLAM Project
roslaunch home_service_project home_service_project_world.launch
    world_file:=/home/nvidia/catkin_ws/src/
    home_service_project/worlds/U.world
roslaunch home_service_project gmapping.launch
roslaunch turtlebot_interactive_markers
   interactive_markers.launch
```

The *home_service_project_world.launch* launch file launches gazebo in the U world and places the robot at home position. It also launches RViz. It is possible to launch RViz as a separate step but there is no advantage to doing so.

The *gmapping.launch* script that was developed in prior projects was used to do the mapping.

Turtlebot interactive markers were used rather than keyboard teleop to move the robot as it is so much more convenient. In order to comply with the project requirements keyboard teleop is a command line option.

This figure shows *test_slam* in action:

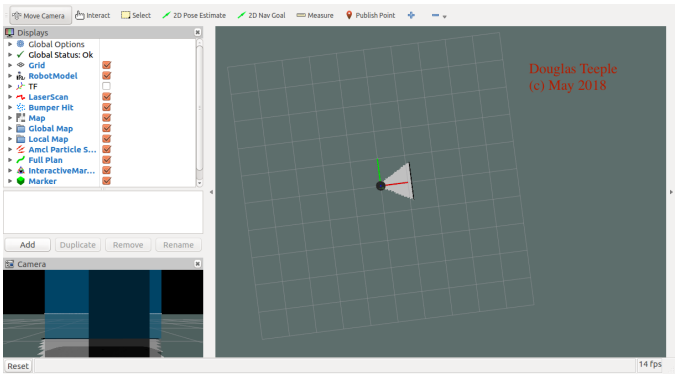

Figure 2. Test SLAM start position.

The *test_slam.sh* script is as follows:

Listing 2. test_slam.sh script

```
#!/bin/bash
##############################################
#
# Udacity Term 2 Home Service Project launch script
#          Douglas Teeple April 2018
#
# Test SLAM operation by using interactive markers or
# teleop_keyboard to create a map.
#
##############################################

. ./colors.sh    # enable terminal colors

function launch() {
        if [[ "$2" != "" ]]
        then
                sleep $2
        fi
        echo -e ${GREEN}"$1"${NC}
        xterm -e "$1" &
}

echo -e ${BLUE}"Testing SLAM Project"${NC}
source /opt/ros/kinetic/setup.bash
source devel/setup.bash

project=home_service_project
ws=$HOME/catkin_ws/
world=U
world_file=${ws}/src/${project}/worlds/${world}.world
map_file=${ws}/src/${project}/maps/${world}.yaml
teleop=interactivemarkers

for arg in $*
do
        case ${arg} in
                teleop=*) teleop=${arg#teleop=};;
                world=*)        world=${arg#world=};
                                world_file=${ws}/src/${project}
                    /worlds/${world}.world;
                                map_file=${ws}/src/${project}
                    /maps/${world}.yaml;
                                mapname=${world};;
                *)      echo -e ${MAGENTA}"Unknown argument:
        ${arg}"${NC};
                        echo -e ${GREEN}"Usage: $(basename $0):
        [teleop=keyboard|interactive]"${NC};
                        ;;
        esac
done

launch "roslaunch ${project} ${project}_world.launch
        world_file:=${world_file}"
launch "roslaunch ${project} gmapping.launch " 5

if [[ "${teleop}" == "keyboard" ]]
then
        launch "roslaunch turtlebot_teleop
        keyboard_teleop.launch" 5
else
        launch "roslaunch turtlebot_interactive_markers
        interactive_markers.launch" 5
fi
```

The script supports colorized terminal output. It has a "launch" function which optionally delays launch by the number of seconds of argument 2, prints the command about to be executed and then launches the command given as argument 1 in a new xterm.

This figure shows test_slam:

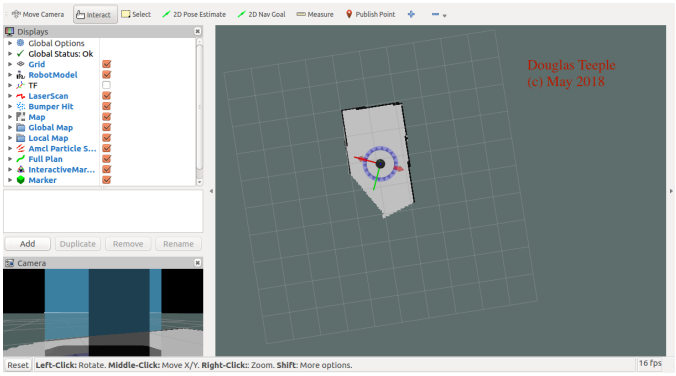

Figure 3. Test SLAM in operation.

The interactive markers are clearly visible around the robot. The robot is moved and rotated around the environment by dragging the interactive markers.

5.2 Wall Follower

The wall follower node autonomously maps environment in order to automate map creation.

Listing 3. test_slam.sh script

```
nvidia@tegra-ubuntu:~/catkin_ws$ ./wall_follower.sh
Starting WallFlower Project
roslaunch home_service_project home_service_project_world.launch
        world_file:=/home/nvidia/catkin_ws/src/
    home_service_project/worlds/U.world
roslaunch home_service_project gmapping.launch
rosrun wall_follower wall_follower
```

This figure shows the robot at the start position:

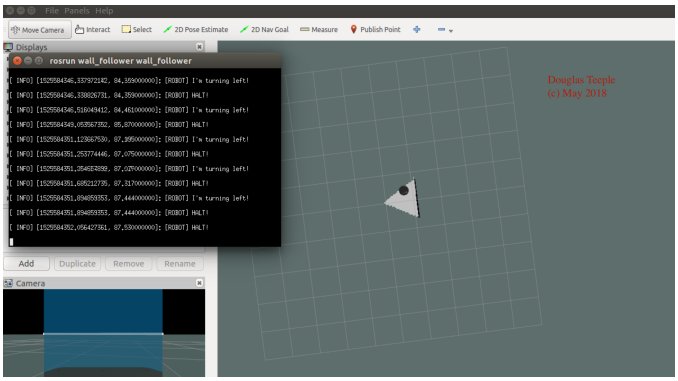

Figure 4. Wall Follower Start Position.

The *wall_follower node* written in C++ autonomously moves the robot and a *wall_follower.sh* script file launches the node to autonomously perform SLAM.

The wall_follower node is a C++ program largely as given in the project description. It subscribes to laser *scan* messages and uses the simple wall following algorithm as given to map the world. It moves the robot by publishing poses to the *cmd_vel_mux/input/navi* topic. The sources to this node reside in the git repository: https://github.com/douglasteeple/HomeServiceRobot.

The algorithm only works for cleanly enclosed worlds. If there is an opening, for example, the robot can wander without ever reaching closure.

Listing 4. wall_follower.sh script

```
#!/bin/bash
#########################################################
#
# Udacity Term 2 Home Service Project launch script
#          Douglas Teeple April 2018
#
# Automate the process to let the robot follow the walls
# and autonomously map the environment while avoiding obstacles.
#
#########################################################

. ./colors.sh   # enable terminal colors

function launch() {
        if [[ "$2" != "" ]]
        then
                sleep $2
        fi
        echo -e ${GREEN}"$1"${NC}
        xterm -e "$1" &
}

echo -e ${BLUE}"Starting WallFlower Project"${NC}
source /opt/ros/kinetic/setup.bash
source devel/setup.bash

source /opt/ros/kinetic/setup.bash
source devel/setup.bash

world=U
project=home_service_project
ws=$HOME/catkin_ws/
world_file=${ws}/src/${project}/worlds/${world}.world
mapname=${world}

for arg in $*
do
        case ${arg} in
                map=*)  mapname=${arg#map=};
                        echo -e ${GREEN}"Saving map ${mapname}..."${NC};
                        launch "rosrun map_server map_saver
----------------f ${ws}/src/${project}/maps/${mapname}";
                        exit;
                        ;;
                world=*)        world=${arg#world=};
                                world_file=${ws}/src/${project}
                    /worlds/${world}.world;
                                map_file=${ws}/src/${project}
                    /maps/${world}.yaml;
                                mapname=${world};;
                *)      echo -e ${MAGENTA}"Unknown argument: ${arg}"${NC};
                        echo -e ${GREEN}"Usage: $(basename $0):
----------------[teleop=keyboard|interactive]"${NC};
                        ;;
        esac
done

launch "roslaunch ${project} ${project}_world.launch
--------world_file:=${world_file}"
launch "roslaunch ${project} gmapping.launch " 5
launch "rosrun wall_follower wall_follower" 10
```

This script launches gazebo and RViz as before and adds a call to run the *wall_follower* node.

This figure shows the robot as it maps the world:

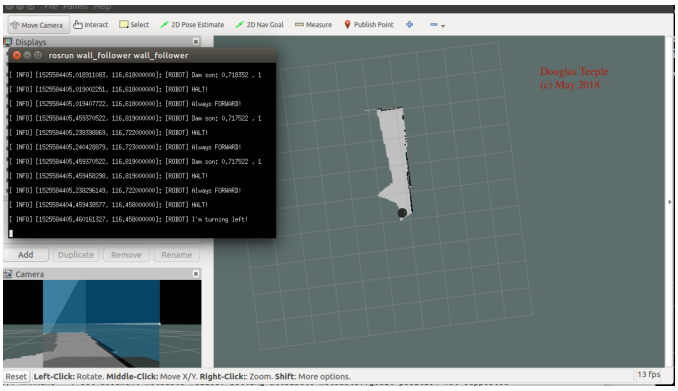

Figure 5. Wall Follower Mapping the World.

The *wall_follower script* has an option to save a map of the environment using the mapping service.

Both a *pgm* and *yaml* file are saved in the worlds directory.

This figure shows the wall_follower map:

The *U.yaml* file created by the map_server:

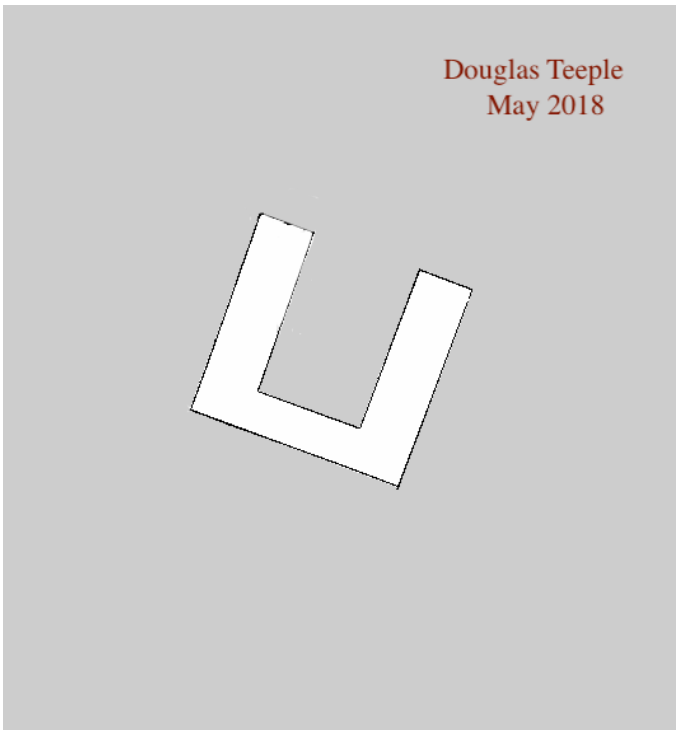

Figure 6. The U map created by wall_follower.

```
image: /home/nvidia/catkin_ws/src/
home_service_project/maps/U.pgm
resolution: 0.050000
origin: [-50.000000, -50.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

6 NAVIGATION

This step manually tests navigation of the robot inside the environment. I used the *2D Nav Goal* in Rviz to command your robot to navigate to two different locations.

```
nvidia@tegra-ubuntu:~/catkin_ws$ ./test_navigation.sh
Starting Navigation Project
Manually point out to two different 2D Pose goals,
one at a time, and direct your robot to reach them
and orient itself with respect to them.
roslaunch home_service_project home_service_project_world.launch
world_file:=/home/nvidia/catkin_ws/src/
home_service_project/worlds/U.world
roslaunch turtlebot_gazebo amcl_demo.launch
map_file:=/home/nvidia/catkin_ws/src/
home_service_project/maps/U.yaml 3d_sensor:=kinect
roslaunch turtlebot_interactive_markers
interactive_markers.launch
```

This figure shows the test_navigation start position in RViz:

The *test_navigation.sh* script file launches the robot, *AMCL* for localization and *teleop* to manually test navigation.

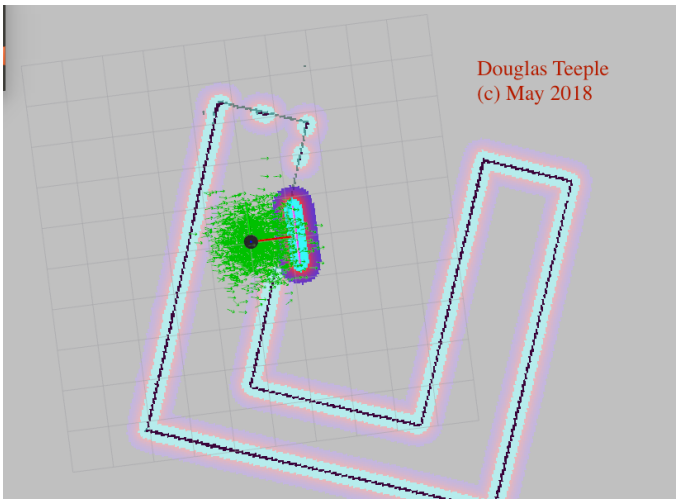

Figure 7. Test Navigation start position in RViz.

Listing 5. test_navigation.sh script

```
#!/bin/bash
##############################################################################
#
# Udacity Term 2 Home Service Project launch script
#           Douglas Teeple April 2018
#
# Manually point out to two different goals, one at a time,
# and direct your robot to reach them and orient itself with
# respect to them.
##############################################################################

. ./colors.sh    # enable terminal colors

function launch() {
        if [[ "$2" != "" ]]
        then
                sleep $2
        fi
        echo -e ${GREEN}"$1"${NC}
        xterm -e "$1" &
}

echo -e ${BLUE}"Starting Navigation Project"${NC}
echo -e ${BLUE}"Manually point out to two different 2D Pose goals,"${NC}
echo -e ${BLUE}"one at a time, and direct your robot to reach them"${NC}
echo -e ${BLUE}"and orient itself with respect to them."${NC}

project=home_service_project
ws=$HOME/catkin_ws/
world=U
world_file=${ws}/src/${project}/worlds/${world}.world
map_file=${ws}/src/${project}/maps/${world}.yaml

for arg in $*
do
        case ${arg} in
                world=*)        world=${arg#world=};
                                world_file=${ws}/src/${project}
                        /worlds/${world}.world;
                                map_file=${ws}/src/${project}
                        /maps/${world}.yaml;
                                mapname=${world};;
                *)      echo -e ${MAGENTA}"Unknown argument:
                ${arg}"${NC};
                        echo -e ${GREEN}"Usage: $(basename $0):
                [teleop=keyboard|interactive]"${NC};
                ;;
        esac
done

source /opt/ros/kinetic/setup.bash
source devel/setup.bash

launch "roslaunch ${project} ${project}_world.launch
        world_file:=${world_file}"
launch "roslaunch turtlebot_gazebo amcl_demo.launch
        map_file:=${map_file} 3d_sensor:=kinect" 5
launch "roslaunch turtlebot_interactive_markers
        interactive_markers.launch" 5
```

This figure shows *test_navigation* at goal 1:

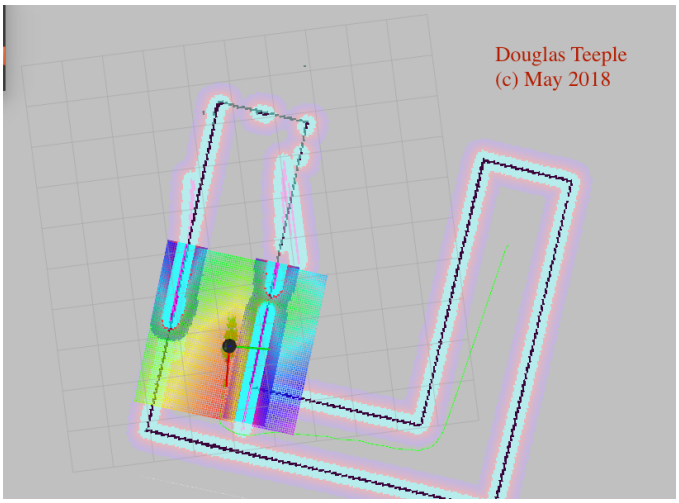

Figure 8. Test Navigation goal one position in RViz.

This figure shows *test_navigation* at goal 2:

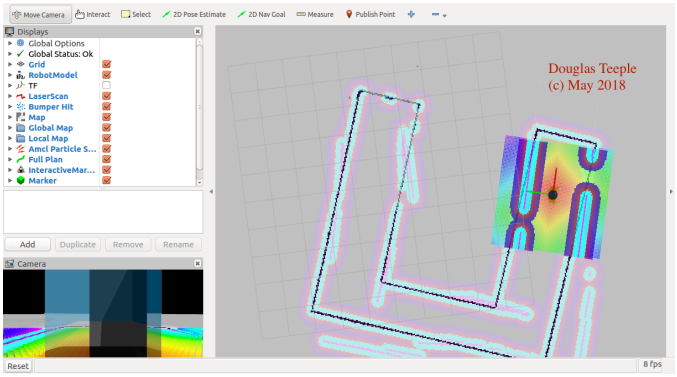

Figure 9. Test Navigation goal two position in RViz.

The next step is to create a node called *pick_objects* that communicates with the ROS navigation stack and autonomously send two goals for the robot to reach.

```
nvidia@tegra-ubuntu:~/catkin_ws$ ./pick_objects.sh
Starting Pick Objects Project
roslaunch home_service_project home_service_project_world.launch
world_file:=/home/nvidia/catkin_ws/src/
home_service_project/worlds/U.world
roslaunch turtlebot_gazebo amcl_demo.launch
map_file:=/home/nvidia/catkin_ws/src/
home_service_project/maps/U.yaml
3d_sensor:=kinect
roslaunch turtlebot_interactive_markers
interactive_markers.launch
rosrun pick_objects pick_objects
```

Execution of the *pick_objects script* launches the U world and AMCL to do localization, launches *interactive_markers teleop* and finally launches the *pick_objects* node.

This figure shows *pick_objects* at goal 1:

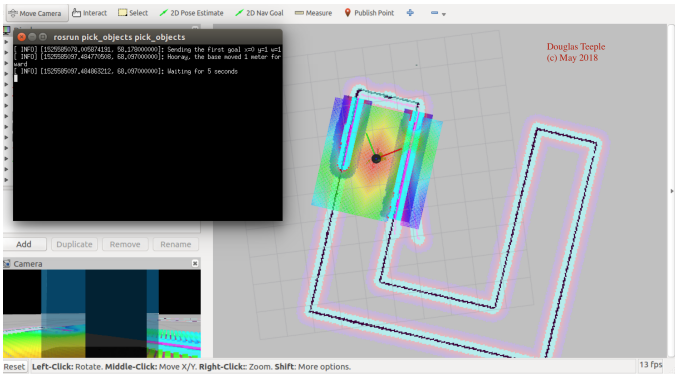

Figure 10. Pick Objects goal 1

The robot travels to the desired pickup zone, displays a message that it reached its destination, waits 5 seconds, and then travels to the desired drop off zone, and displays a message that it reached the drop off zone.

This figure shows *pick_objects* at goal 2:

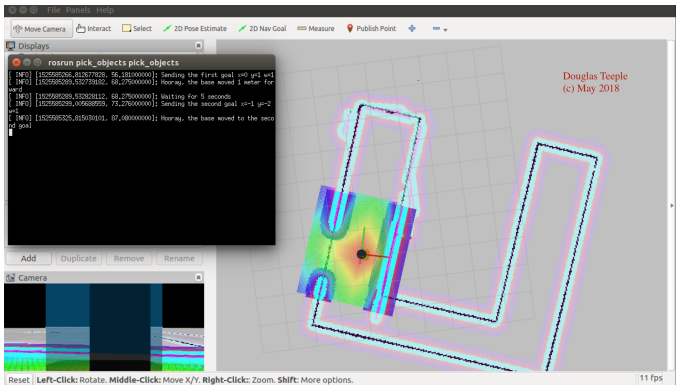

Figure 11. Pick Objects goal two.

The *pick_objects.sh* file sends multiple goals for the robot to reach:

Listing 6. pick_objects.sh script

```
#!/bin/bash
##############################################################
#
# Udacity Term 2 Home Service Project launch script
#          Douglas Teeple April 2018
#
# Navigate to two goals.
# The first goal should be your desired pickup goal and the second goal
# should be your desired drop off goal. The robot has to travel to the
# desired pickup zone, display a message that it reached its destination,
# wait 5 seconds, travel to the desired drop off zone, and display a
# message that it reached the drop off zone.
#
##############################################################

. ./colors.sh   # enable terminal colors

function launch() {
        if [[ "$2" != "" ]]
        then
                sleep $2
        fi
        echo -e ${GREEN}"$1"${NC}
        xterm -e "$1" &
}

echo -e ${BLUE}"Starting Pick Objects Project"${NC}

source /opt/ros/kinetic/setup.bash
source devel/setup.bash

project=home_service_project
ws=$HOME/catkin_ws/
world=U
world_file=${ws}/src/${project}/worlds/${world}.world
map_file=${ws}/src/${project}/maps/${world}.yaml
teleop=interactivemarkers

for arg in $*
do
        case ${arg} in
                teleop=*) teleop=${arg#teleop=};;
                world=*)        world=${arg#world=};
                                world_file=${ws}/src/${project}
                    /worlds/${world}.world;
                                map_file=${ws}/src/${project}
                    /maps/${world}.yaml;
                                mapname=${world};;
                *)      echo -e ${MAGENTA}"Unknown argument:
                ${arg}"${NC};
                        echo -e ${GREEN}"Usage: $(basename $0):
                [teleop=keyboard|interactive]"${NC};
                        ;;
        esac
done

launch "roslaunch ${project} ${project}_world.launch
        world_file:=${world_file}"
launch "roslaunch turtlebot_gazebo amcl_demo.launch
        map_file:=${map_file} 3d_sensor:=kinect" 5
if [[ "${teleop}" == "keyboard" ]]
then
        launch "roslaunch turtlebot_teleop
        keyboard_teleop.launch" 5
else
        launch "roslaunch turtlebot_interactive_markers
        interactive_markers.launch" 5
fi
launch "rosrun pick_objects pick_objects"
```

7 MARKERS

This step models a virtual object with markers in RViz. I chose a coffee cup as the virtual object.

The *add_marker.sh* script launches the *add_markers* node which publishes a marker to RViz initially at the pickup zone. After 5 seconds it is hidden, then after another 5 seconds it appears at the drop off zone.

The script execution log shows the launch steps:

```
vidia@tegra-ubuntu:~/catkin_ws$ ./add_markers.sh
Starting Add Markers Project
roslaunch home_service_project home_service_project_world.launch
world_file:=/home/nvidia/catkin_ws/src/
home_service_project/worlds/U.world
roslaunch home_service_project amcl.launch
roslaunch turtlebot_interactive_markers  interactive_markers.launch
rosrun add_markers add_markers
```

This figure shows the robot at the start point:

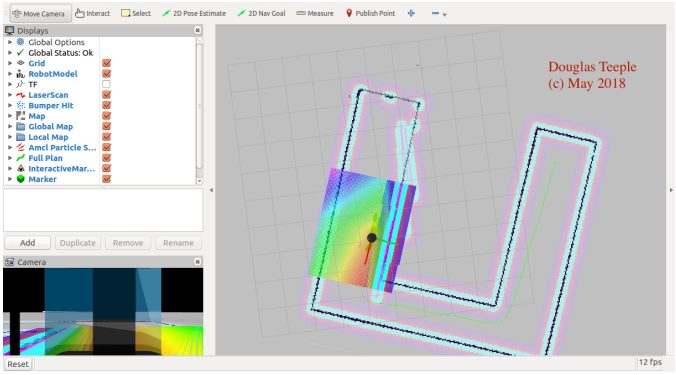

Figure 12. Add Markers Start.

The *add_markers.sh* bash script is as follows:

Listing 7. add_markers.sh script

```
#!/bin/bash
##############################################################
#
# Udacity Term 2 Home Service Project launch script
#          Douglas Teeple April 2018
#
# Navigate to two goals and move a marker from the pickup goal to
# the dropoff goal.
# The first goal should be your desired pickup goal and the second goal
# should be your desired drop off goal. The robot has to travel to the
# desired pickup zone, display a message that it reached its destination,
# wait 5 seconds, travel to the desired drop off zone, and display a
# message that it reached the drop off zone.
#
##############################################################

. ./colors.sh   # enable terminal colors

function launch() {
        if [[ "$2" != "" ]]
        then
                sleep $2
        fi
        echo -e ${GREEN}"$1"${NC}
        if [[ "$2" == "0" ]]
        then
                xterm -e "$1"
        else
                xterm -e "$1" &
        fi
}

echo -e ${BLUE}"Starting Add Markers Project"${NC}

source /opt/ros/kinetic/setup.bash
source devel/setup.bash

project=home_service_project
ws=$HOME/catkin_ws/
world=U
world_file=${ws}/src/${project}/worlds/${world}.world
map_file=${ws}/src/${project}/maps/${world}.yaml
teleop=interactivemarkers

for arg in $*
do
        case ${arg} in
                teleop=*) teleop=${arg#teleop=};;
                world=*)        world=${arg#world=};
                                world_file=${ws}/src/${project}
                    /worlds/${world}.world;
                                map_file=${ws}/src/${project}
                    /maps/${world}.yaml;
                                mapname=${world};;
                *)      echo -e ${MAGENTA}"Unknown argument:
________________${arg}"${NC};
                        echo -e ${GREEN}"Usage: $(basename $0):
________________[teleop=keyboard|interactive]"${NC};
                        ;;
        esac
done

launch "roslaunch ${project} ${project}_world.launch
________world_file:=${world_file}"
launch "roslaunch ${project} amcl.launch " 5

if [[ "${teleop}" == "keyboard" ]]
then
        launch "roslaunch turtlebot_teleop
________keyboard_teleop.launch" 5
else
        launch "roslaunch turtlebot_interactive_markers
________interactive_markers.launch" 5
fi
launch "rosrun add_markers add_markers"
```

The *add_markers.sh* script follows previous launch steps then runs the add_markers node.

The *add_markers* node is a C++ program *add_markers.cpp*. It simplifies multiple goals using a *doSendGoalsAndWait* function which accepts a vector of goals. The goals are then called as:

Listing 8. Send a Vector of Poses

```
// send a vector of goals
std::vector< geometry_msgs::Pose > poses;
poses.push_back(newPose(0.0, 1.0));
poses.push_back(newPose(6.5, 0.0));
poses.push_back(newPose(0.0, 0.0));
doSendGoalsAndWait(ac, marker, marker_pub, poses);
```

The entire program is quite large and can be found in the git repository.

This figure shows the gold box marker at the goal:

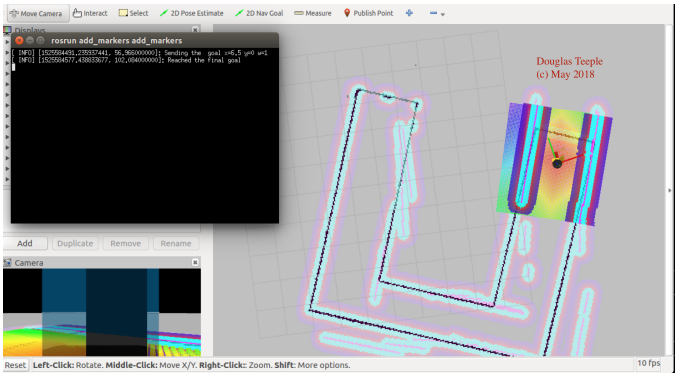

Figure 13. Add Markers showing the gold box marker at goal.

8 HOME SERVICE

In the final step the *add_markers* node is modified to subscribe to robot odometry, and publish the markers that follow the robot position. The new C++ program *home_service.cpp* was created. It initially shows the marker at the pickup zone, then hides the marker once the robot reaches the pickup zone. The pickup is simulated by a 5 second wait. Then the robot "carries" the marker to the drop off zone.

The *home_service.sh* script launches the environment and the *home_service* node. It executes as follows:

```
nvidia@tegra-ubuntu:~/catkin_ws$ ./home_service.sh
Starting Home Service Project
roslaunch home_service_project home_service_project_world.launch
world_file:=/home/nvidia/catkin_ws/src/
home_service_project/worlds/U.world
roslaunch turtlebot_gazebo amcl_demo.launch
map_file:=/home/nvidia/catkin_ws/src/
home_service_project/maps/U.yaml
    3d_sensor:=kinect
roslaunch turtlebot_interactive_markers
interactive_markers.launch
rosrun home_service_project home_service
```

It launches the environment as before and then calls the *home_service* node.

The C++ source file for the *home_service* node is quite long so only cogent pieces are described. First, the main function executes the basic program steps:

Listing 9. main function

```
        // Initialize the simple_navigation_goals node
        ros::init(argc, argv, "home_service");

        // Create a marker
        coffeecup = createMarker(&marker_pub);

#ifdef USE_ODOM
        // Create odometer subscriber
        ros::NodeHandle nodehandle;
        ros::Subscriber sub = nodehandle.subscribe("odom", 1000, odomCallback);
#endif
        //tell the action client that we want to spin a thread by default
        MoveBaseClient ac("move_base", true);

        // Wait 5 sec for move_base action server to come up
        while(!ac.waitForServer(ros::Duration(5.0))){
                ROS_INFO("Waiting for the move_base action server to come up");
        }

        // wait so the screens can get organized
        ros::Duration(10.0).sleep(); // sleep for 10 seconds

        woohoo = createMarker(&marker_pub, "Woohoo! Coffee is Served", 0.5);

        // show marker at pick up point for 5 seconds...
        showMarkerAt(-0.5, -1.0, coffeecup, 5.0);

        // send a vector of pairs of goals and whether to show marker
        std::vector< std::pair<geometry_msgs::Pose, bool>> posepairs;
        // move to the pickup point without the marker
        posepairs.push_back({newPose(-0.5, -1.0),false});
        // move to the dropoff point showing the marker
        posepairs.push_back({newPose(6.5, 0.0),true});
        // off we go...
        doSendGoalsAndWait(ac, posepairs, coffeecup);
        // Woohoo!
        showMarkerAt(6.5, 0.0, woohoo);

        // spin so the messages remain on the terminal screen...
        ros::spin();
```

The basic steps are:

- Initialize the *home_service* node.
- Create the Coffee Cup payload marker.
- if the define *USE_ODOM* is true, subscribe the odometer messages. Note that I had trouble (as is described in detail in the discussion section below) in smooth operation of payload carrying using the method suggested in the Project Description. Instead I opted to attach the payload to the *base_frame* of the robot to implement following.
- Create a move base client.
- Wait for the move_base action server to launch.
- Create the *"success"* text marker.
- Show the Coffee Cup marker at the pick up point for 5 seconds.
- Create a vector of poses of the goals in the project. The two goals are: 1) Move to the pick up point (not carrying the Coffee Cup marker). 2) Move to the drop off point carrying the marker.
- Send the goals.
- Display the success text marker.

The *newPose* function initializes a pose data structure:

Listing 10. newPose function

```
const geometry_msgs::Pose newPose(float px, float py,
        float pz=0, float qx=0, float qy=0,
    float qz=0, float qw=1) {
        geometry_msgs::Pose pose;
        pose.position.x = px;
        pose.position.y = py;
        pose.position.z = pz;
        pose.orientation.x = qx;
        pose.orientation.y = qy;
        pose.orientation.z = qz;
        pose.orientation.w = qw;
        return pose;
}
```

The *createMarker* function creates either a mesh (Coffee Cup) or a text marker (success text):

Listing 11. createMarker function

```
visualization_msgs::Marker createMarker(
        ros::Publisher *marker_pub,
    const char *text=nullptr,
    float howbig=0.5) {
        ros::NodeHandle nodehandle;
        if (marker_pub && !*marker_pub) *marker_pub =
        nodehandle.advertise<visualization_msgs::Marker>
                ("visualization_marker", 1);
        visualization_msgs::Marker marker;
        // Set the frame ID and timestamp.
        marker.header.frame_id = "map";//"base_footprint";
        marker.header.stamp = ros::Time::now();
        // Set the namespace and id for this marker.
        marker.ns = "basic_shapes";
        marker.id = 0;
        // Set the marker type.
        if (text) {
                marker.type = visualization_msgs::Marker::TEXT_VIEW_FACING;
                marker.text = std::string(text);
                marker.scale.x = howbig;
                marker.scale.y = howbig;
                marker.scale.z = howbig;
                // Set the color
                // Dodger blue: 0.118, 0.565, 1.000
                // Lawn Green: 0.486 0.988 0
                marker.color.r = 0.486;
                marker.color.g = 0.988;
                marker.color.b = 0.0;
                marker.color.a = 1.0;
        } else {
                marker.type = visualization_msgs::Marker::MESH_RESOURCE;
                marker.mesh_resource =
      "package://home_service_project/meshes/CoffeeCup.dae";
                // Set the scale of the marker
                marker.scale.x = 0.125;
                marker.scale.y = 0.125;
                marker.scale.z = 0.125;
                // Set the color
                // Gold: 1.000, 0.843, 0.000
                marker.color.r = 1.0;
                marker.color.g = 0.843;
                marker.color.b = 0.0;
                marker.color.a = 1.0;
        }
        // Set the marker action.
        marker.action = visualization_msgs::Marker::ADD;
        // Set the pose of the marker.
        marker.pose.position.x = 0;
        marker.pose.position.y = 0;
        marker.pose.position.z = 0;
        marker.pose.orientation.x = 0.0;
        marker.pose.orientation.y = 0.0;
        marker.pose.orientation.z = 0.0;
        marker.pose.orientation.w = 1.0;
        marker.lifetime = ros::Duration(100000);
        return marker;
}
```

The *doSendGoalsAndWait* function is the main workhorse function. It received a vector of goal poses and moves the robot to each goal in turn. The boolean flag in the vector indicates whether the payload should be shown or not.

Listing 12. doSendGoalsAndWait function

```
bool doSendGoalsAndWait(MoveBaseClient &ac,
        std::vector< std::pair<geometry_msgs::Pose,
        bool>>
    const &posevector,
    visualization_msgs::Marker &marker) {
        move_base_msgs::MoveBaseGoal goal;
        while (marker_pub.getNumSubscribers() < 1) {
                if (!ros::ok()) {
                        return 0;
                }
                ROS_WARN_ONCE("Please create a "
                "subscriber to the marker");
                sleep(1);
        }
        // set up the frame parameters
        goal.target_pose.header.frame_id = "map";
        goal.target_pose.header.stamp =
        ros::Time::now();
        // Define a position and
    // orientation for the robot to reach
        size_t count = posevector.size();
        for (const auto &posepair : posevector) {
                geometry_msgs::Pose pose = posepair.first;
                active = posepair.second;
                marker.pose = goal.target_pose.pose = pose;
                marker.action =
                visualization_msgs::Marker::ADD;
                count--;
                // Send the goal position and
        // orientation for the robot to reach
                ROS_INFO("Sending the goal x=%g y=%g w=%g",
        goal.target_pose.pose.position.x,
        goal.target_pose.pose.position.y,
        goal.target_pose.pose.orientation.w);
                ac.sendGoal(goal,
                &doneCb, &activeCb, &feedbackCb);
                // Wait 60 seconds for the results
                bool finished_before_timeout =
                ac.waitForResult(ros::Duration(60.0));
                // Check if the robot reached its goal
                if ((ac.getState() ==
      actionlib::SimpleClientGoalState::SUCCEEDED)) {
                        ROS_INFO("Reached the %sgoal",
                (count==0?"final ":""));
                        marker.action =
             visualization_msgs::Marker::ADD;
                        if (active) marker_pub.publish(marker);
                        const int wait_time = 5;
                        ROS_INFO("Waiting for %d seconds",
                wait_time);
                        int msecs = wait_time*1000;
                        while (msecs--) {
                          // defer to other callbacks
                          ros::spinOnce();
                          ros::getGlobalCallbackQueue()
                ->callAvailable(
                        ros::WallDuration(0.001));
                        }
                } else {
                        ROS_WARN("Failed move to the goal");
                }
        }
}
```

Listing 13. sendGoal Callback Functions

```
void doneCb(const actionlib::SimpleClientGoalState& state,
        const MoveBaseResultConstPtr& result)
{
  ROS_INFO("Finished in state [%s]", state.toString().c_str());
  active = false;
}
// Called once when the goal becomes active
void activeCb()
{
        ROS_INFO("Goal just went active");
        //active = true;
        ros::spinOnce();
}
// Called every time feedback is received for the goal
void feedbackCb(const MoveBaseFeedbackConstPtr& feedback)
{
        coffeecup.pose.position =
        feedback->base_position.pose.position;
        coffeecup.pose.position.z += 0.5;
        if (active) marker_pub.publish(coffeecup);
        ros::spinOnce();
}
```

The basic process is an iteration through the goal vector, calling the ROS *sendGoal* function on each goal in the vector and waiting for the goal to be reached. There is also some considerable code calling *spinOnce* in an attempt to defer to the *odom* callback thread. However this method was never very successful and so the defers are not used.

The callback functions from the *sendGoal* method implement various logging featurs. The function *feedbackCb* implements moving the Coffee Cup marker to follow the position of the robot given in the feedback argument. This is the key implementation of carrying the Coffee Cup in simulation.

The home service robot is simulated as follows:

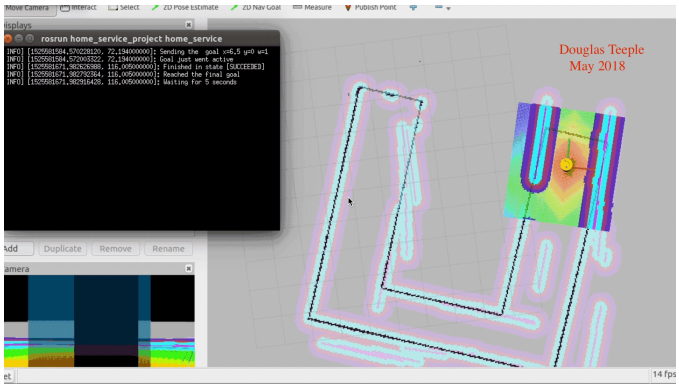

Figure 14. Home Service in Action.

This shows the virtual object that the robot carries - a coffee cup: I used a license-free Coffee Cup mesh *dae* file to

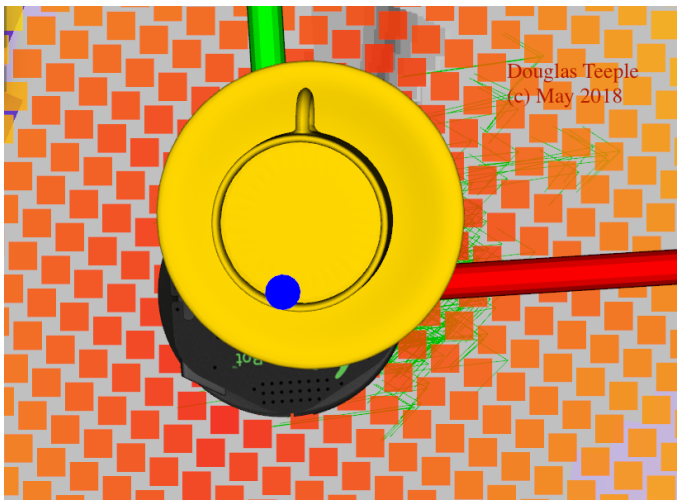

Figure 15. Closeup of the payload.

create the marker.

This figure shows the coffee cup at goal position with the coffee cup:

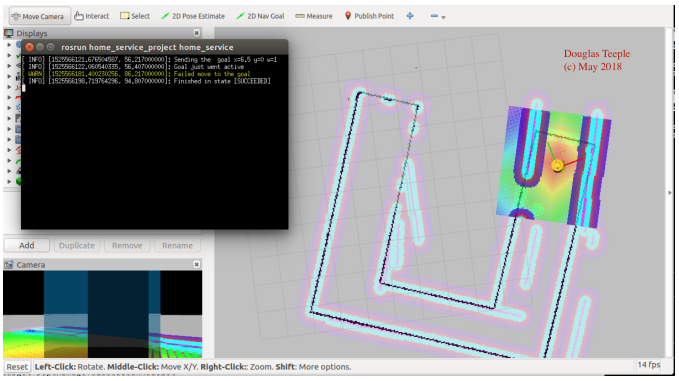

Figure 16. Home Service showing the coffee cup marker at goal.

Coffee delivered!:

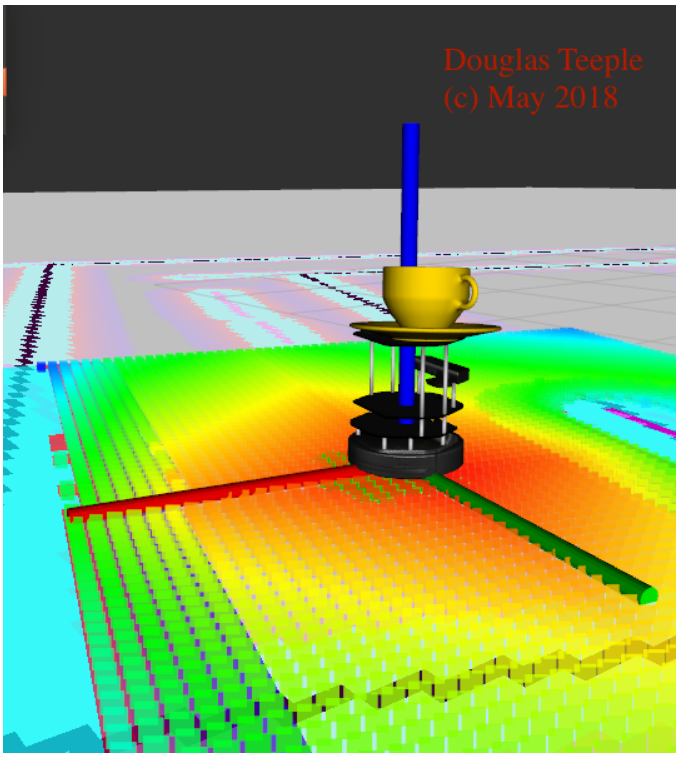

Figure 17. Coffee Delivered!.

The *home_service.sh* is as follows:

Listing 14. home_service.sh script

```
#!/bin/bash
##############################################################
#
# Udacity Term 2 Home Service Project launch script
#          Douglas Teeple April 2018
#
# Navigate to two goals and carry a marker from the pickup goal to
# the dropoff goal.
# The first goal should be your desired pickup goal and the second goal
# should be your desired drop off goal. The robot has to travel to the
# desired pickup zone, display a message that it reached its destination,
# wait 5 seconds, travel to the desired drop off zone, and display a
# message that it reached the drop off zone.
#
##############################################################

. ./colors.sh   # enable terminal colors

function launch() {
        if [[ "$2" != "" ]]
        then
                sleep $2
        fi
        echo -e ${GREEN}"$1"${NC}
        if [[ "$2" == "0" ]]
        then
                xterm -e "$1"
        else
                xterm -e "$1" &
        fi
}

echo -e ${BLUE}"Starting Home Service Project"${NC}

source /opt/ros/kinetic/setup.bash
source devel/setup.bash

project=home_service_project
ws=$HOME/catkin_ws/
world=U
world_file=${ws}/src/${project}/worlds/${world}.world
map_file=${ws}/src/${project}/maps/${world}.yaml
teleop=interactivemarkers

for arg in $*
do
        case ${arg} in
                teleop=*) teleop=${arg#teleop=};;
                world=*)        world=${arg#world=};
                                world_file=${ws}/src/${project}
                    /worlds/${world}.world;
                                map_file=${ws}/src/${project}
                    /maps/${world}.yaml;
                                mapname=${world};;
                *)      echo -e ${MAGENTA}"Unknown argument:
________________${arg}"${NC};
                        echo -e ${GREEN}"Usage: $(basename $0):
________________[teleop=keyboard|interactive]"${NC};
                        ;;
        esac
done

launch "roslaunch ${project} ${project}_world.launch
________world_file:=${world_file}"
launch "roslaunch turtlebot_gazebo amcl_demo.launch
________map_file:=${map_file} 3d_sensor:=kinect" 5

if [[ "${teleop}" == "keyboard" ]]
then
        launch "roslaunch turtlebot_teleop
________keyboard_teleop.launch" 5
else
        launch "roslaunch turtlebot_interactive_markers
________interactive_markers.launch" 5
fi
launch "rosrun ${project} home_service"
```

The project description mentioned that: *"The student should write a home_service.sh file that will run all the nodes in this project"*. I took that to mean that it should build on all the steps in this project to implement the final home_service goals.

9 Supporting Files

9.1 Launch Files

The following launch scripts were created or used:

1) **home_service_project_world.launch**: launches the gazebo with the U world and robot at goal position, then launches RViz.
2) **robot_description.launch**: launches the *joint_state_publisher* and *robot_state_publisher*.
3) **amcl_demo.launch**: Starts the AMCL localization server.
4) **amcl.launch**: Starts a local copy of the AMCL localization server.
5) **gmapping.launch**: Starts the gmapping server.
6) **turtlebot_teleop keyboard_teleop.launch**: launches the teleop service for robot movement.
7) **turtlebot_interactive_markers interactive_markers.launch**: launches interactive marker support for moving the robot.

9.2 Results

The home service robot in its final form is simulated as follows:

- Initially show the marker at the pickup zone.
- Hide the marker once your robot reach the pickup zone.
- Wait 5 seconds to simulate a pickup.
- Show the marker at the drop off zone once your robot reaches it.

Additionally a second text marker declaring success "Woohoo!" was shown upon reaching the goal.

The launch file *"home_service.sh"* launches the various components. NOTE: I made a slight deviation from the verbatim text of the project requirement in that I combined the *pick_objects* node and the *add_markers* node into a single *home_service* node.

An animation of the final *home_service* in operation can be seen here: Home Service Project.

The project successfully met the course requirements of creating a ROS package that is able to launch the robot and have it map the surrounding area with the models. A custom Gazebo world was created and a map for that environment was created as well.

10 Discussion

I modified one aspect of the project to improve the final *"robot carrying virtual object"* step. I found when I used the method of subscribing to the robot odometer to implement movement of the virtual marker, that the *odom* callback was not called often enough as the wait for the robot to reach the goal took most of the available CPU time. I tried various methods of deferring the wait so that the *odom* callback was smooth, but the results were still unsatisfactory. So instead of attaching the virtual marker frame to the map, I attached it to the base of the robot. So, as the robot moved, the coffee cup virtual marker automatically followed. The results were very smooth and this method was adopted.

I also found it very difficult to get a map that was clean enough for the robot to successfully navigate the world. I ended up running the wall_follower node overnight to get the level of detail necessary.

I had trouble with AMCL local mapping being out of sync with the global map. As result the local map would close off corners and prevent path planning from finding a route to the goal. I found that only after creating a very high quality map could the robot complete the path plan.

I tried to map the *kitchen_dining.world* to see how the algorithm would work. This figure shows the start of wall following in gazebo:

Figure 18. Wall Following the the Kitchen_dining world.

It did poorly, the robot is cornered and confused by objects in the room:

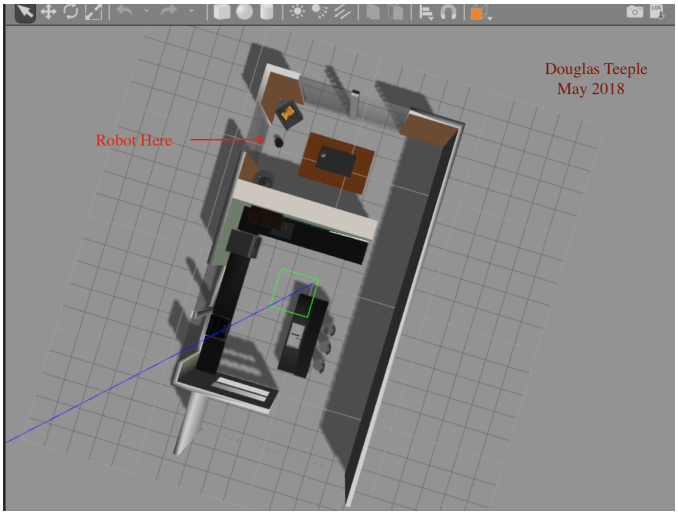

Figure 19. Here's Trouble!

And it got stuck at the open door, so the map doesn't look good:

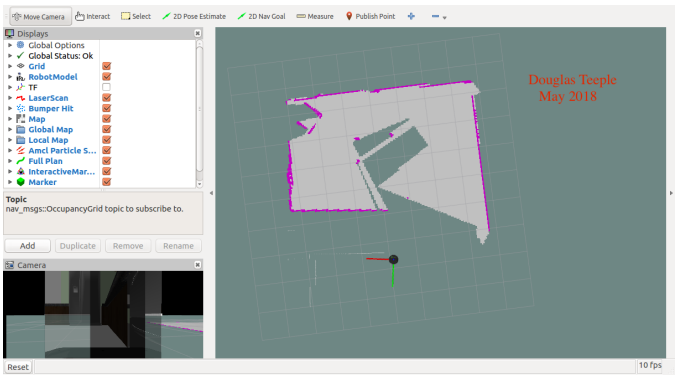

Figure 20. Not a Good Map

Clearly a better mapping algorithm is needed for more complex world that contain objects and open doors.

11 Conclusion / Future work

The Jetson TX2 is an excellent platform to implement this project. It has enough computing power to smoothly implement the robot motions. Using this platform I was able to successfully complete all aspects of the project. In the future it would be interesting to put the Jetson board on an actual *roomba* or *turlebot* robot and have it move through a real environment.

The wall following step was very weak, and research into better algorithms is in order.