

Implementation of ROS Navigation Stack and Custom Robot Model to Simulate Home Service Robot

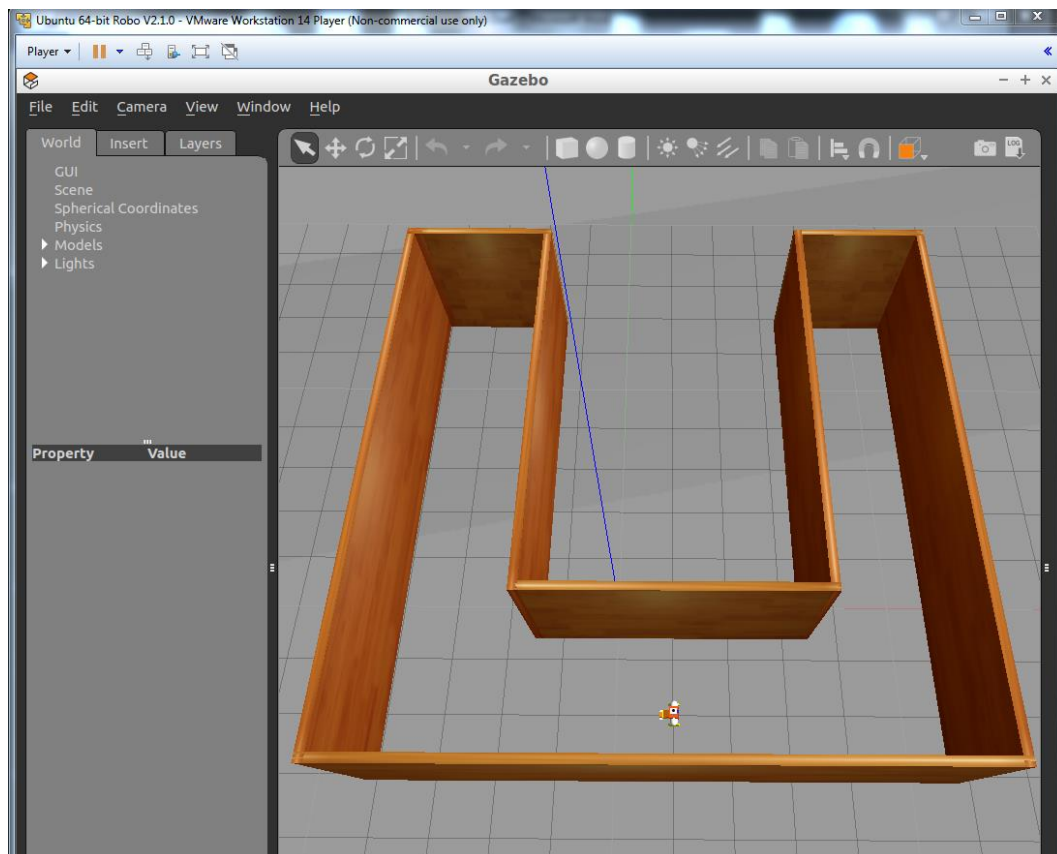
Arnold Kompaniyets

Robotic Software Engineering, Term 2

Project 5

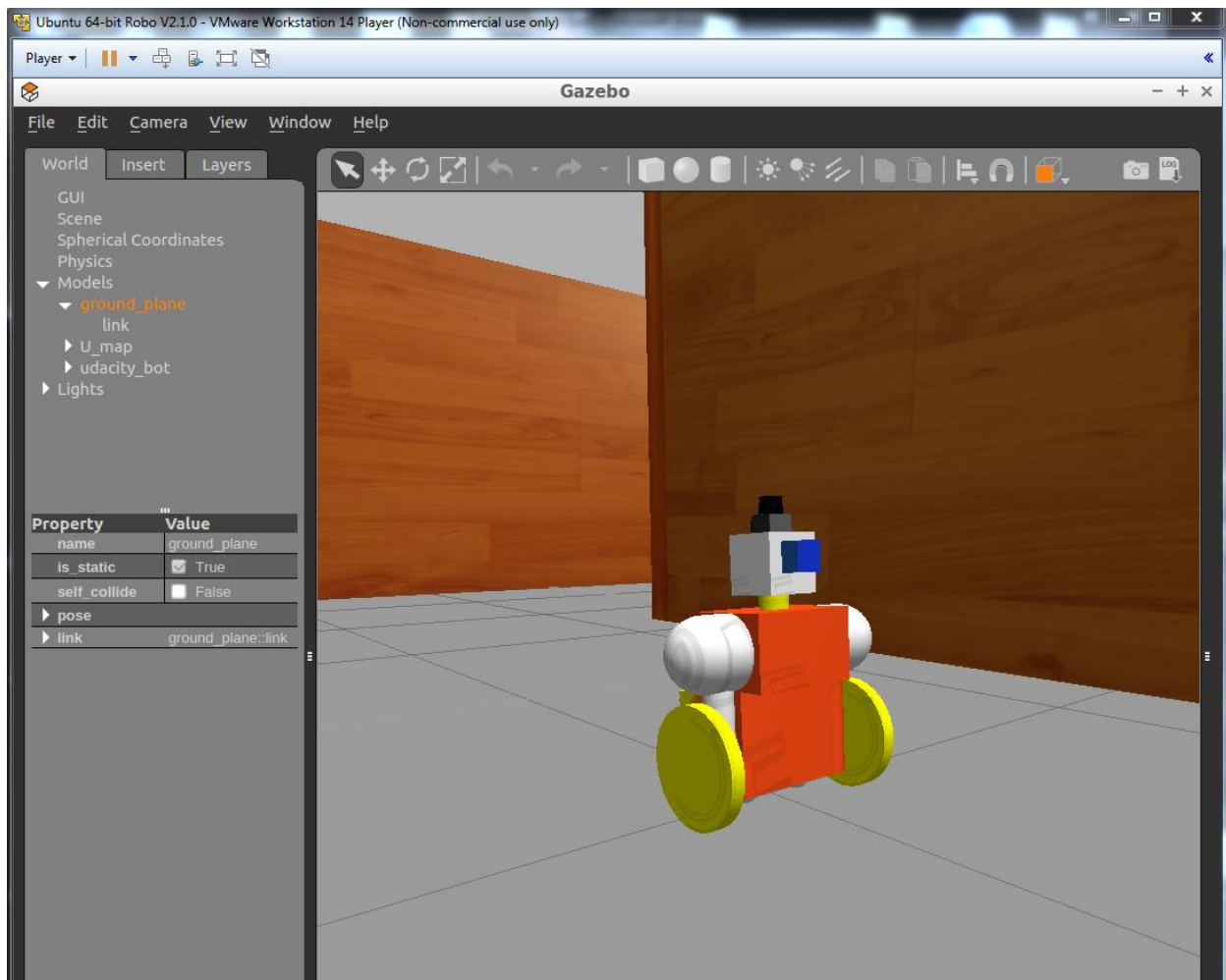
Environment:

The Building Editor tool was used in Gazebo to replicate the Udacity “U” map, akin to the one illustrated in the lessons provided. The measurements detailed in the lessons were also followed, with the goal of closely matching the scale of the intended map. The end result appeared as such in Gazebo, following the application of a wooden texture to the walls:



Robot Model:

The robot model used for this particular project was the custom model created for the localization project in term 2. This custom robot, coined “Gorilla-bot” was structured with a custom body, two wheels alongside its two arms, along with a Hokuyo laser scanner on top of its head. The robot’s corresponding URDF and Gazebo files were placed in the “my_bot” folder. The final Gorilla-bot appeared as such:

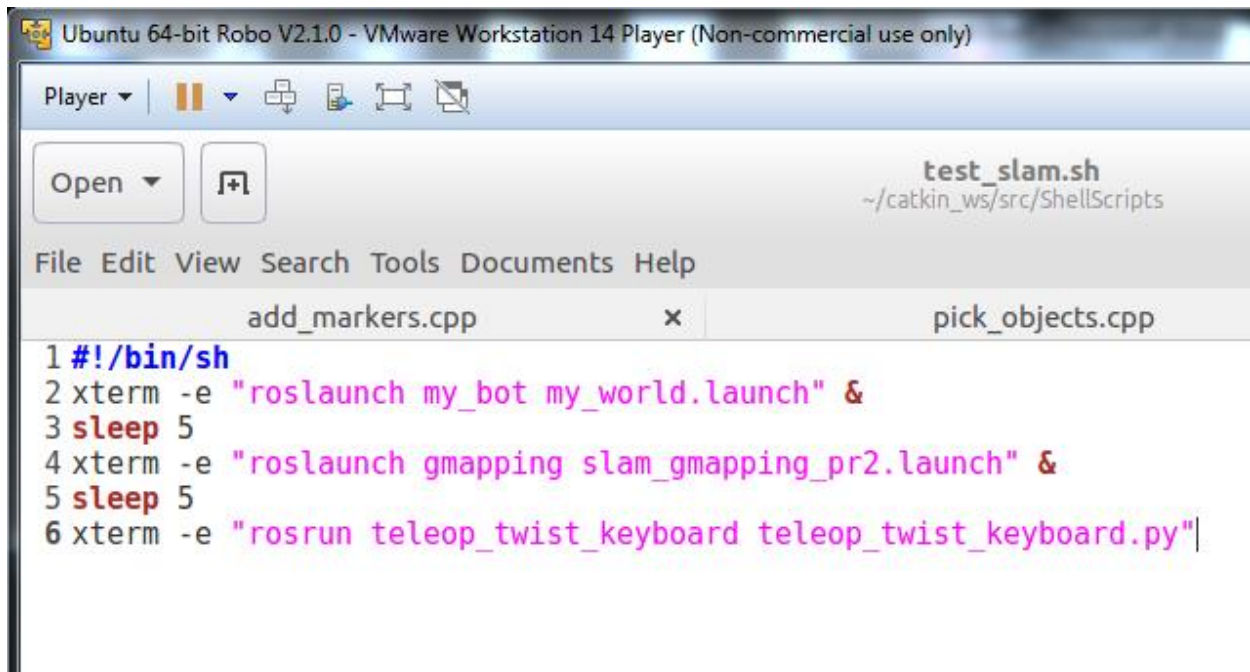


It should be noted that an attempt was made to try and use the TurtleBot package. However, the package was deemed quite difficult to work

with, simply due to the fact that minimal, if any, explanation was provided on how to run the TurtleBot within the project lessons. A list of ROS packages was provided, although many of the required dependencies were rather difficult to properly install and control of the TurtleBot proved far more complex than expected. Specifically, the TurtleBot's movement appeared to run through the "cmd_vel_mux" package, which implemented further packages under the name "kobuki", as well as generally wanting to open up a multitude of nodelets. The fact that the lessons provided made zero mention of all this complexity (and instead made the process seem as easy as running a few terminals) made the entire process entirely frustrating and confusing. After much trouble, even after successfully being able to adjust the TurtleBot's starting position to within the U-map, it refused to accept any movement commands and instead merely slowly turned in place. Therefore, having previously learned how to use keyboard tele-op for a previous project's custom robot model, it was deemed much more convenient to proceed with the custom robot instead.

Mapping:

Following the integration of the custom robot model with the designed Gazebo map, a script file was made to manually test SLAM functionality. The file instantiated three terminals: the first opened the Gazebo and Rviz environments, both of which contained the Gorilla-bot; the second script opened the mapping functionality with the "gmapping" package; the third script opened the "tele-op twist keyboard" package that was added to the ROS Kinetic library. The script appears as such:



Ubuntu 64-bit Robo V2.1.0 - VMware Workstation 14 Player (Non-commercial use only)

Player ▾ | [Icons]

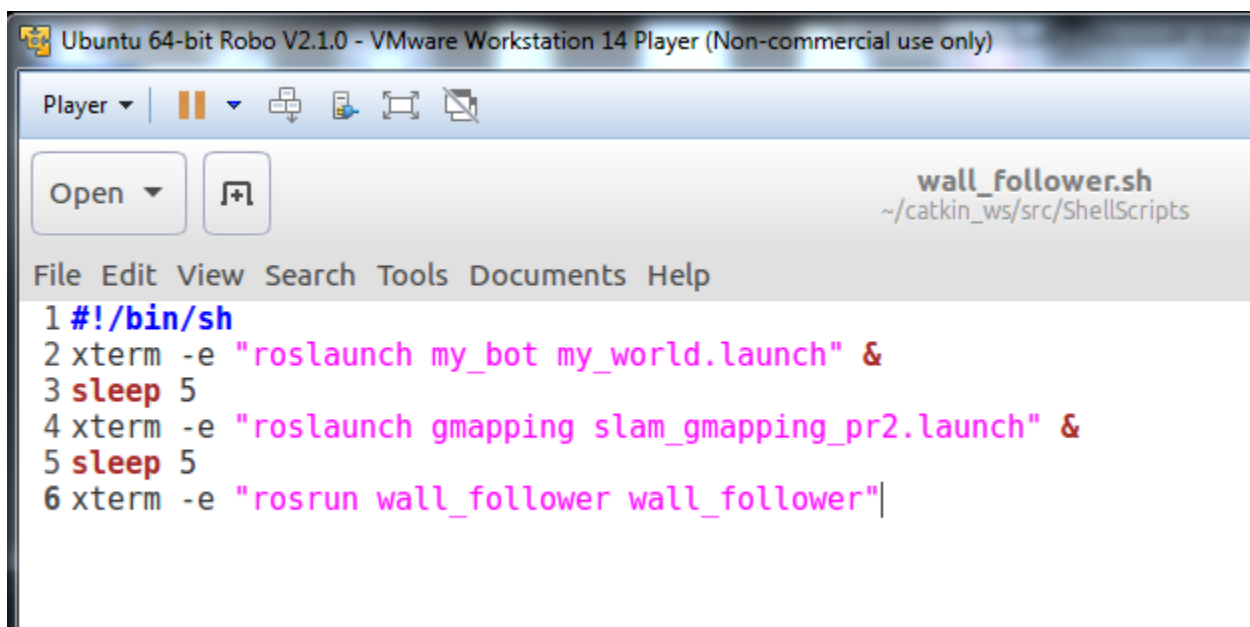
Open ▾ [Icon] **test_slam.sh**
~/catkin_ws/src/ShellScripts

File Edit View Search Tools Documents Help

add_markers.cpp × pick_objects.cpp

```
1 #!/bin/sh
2 xterm -e "roslaunch my_bot my_world.launch" &
3 sleep 5
4 xterm -e "roslaunch gmapping slam_gmapping_pr2.launch" &
5 sleep 5
6 xterm -e "roslaunch teleop_twist_keyboard teleop_twist_keyboard.py"
```

After confirming the functionality of the “gmapping” package, a `wall_follower` node was created to allow the robot to autonomously map its environment. As with the previous task, another shell script was made for this task, which appeared as such:



Ubuntu 64-bit Robo V2.1.0 - VMware Workstation 14 Player (Non-commercial use only)

Player ▾ | [Icons]

Open ▾ [Icon] **wall_follower.sh**
~/catkin_ws/src/ShellScripts

File Edit View Search Tools Documents Help

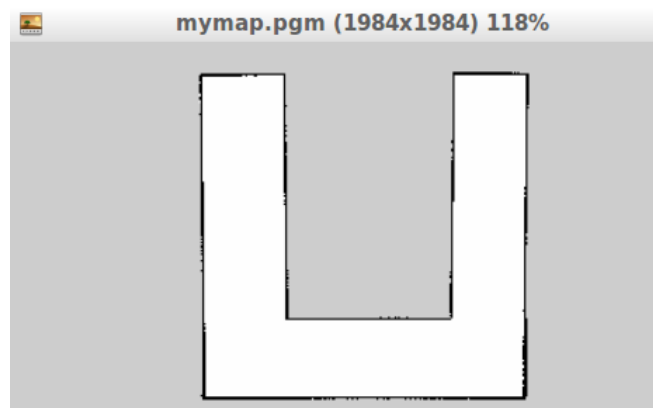
```
1 #!/bin/sh
2 xterm -e "roslaunch my_bot my_world.launch" &
3 sleep 5
4 xterm -e "roslaunch gmapping slam_gmapping_pr2.launch" &
5 sleep 5
6 xterm -e "roslaunch wall_follower wall_follower"
```

The node used instead of the keyboard tele-op was largely kept similar in structure, although simplified in the detail. Instead of leaving in the “following_wall” and “that’s a door” variables, the main logic of the node was narrowed down to the following: if the robot has not crashed, go forward; in the case that the minimum attained laser rangefinder value is below 0.5 meters (signaling proximity to a wall), the robot will turn towards the side that is more open:

```
// Assign movements to a robot that still did not crash
if (!crashed) {
    if (range_min < 0.5) {
        if (left_side >= right_side){robot_move(TURN_LEFT);}
        else {robot_move(TURN_RIGHT);}
    }
}
```

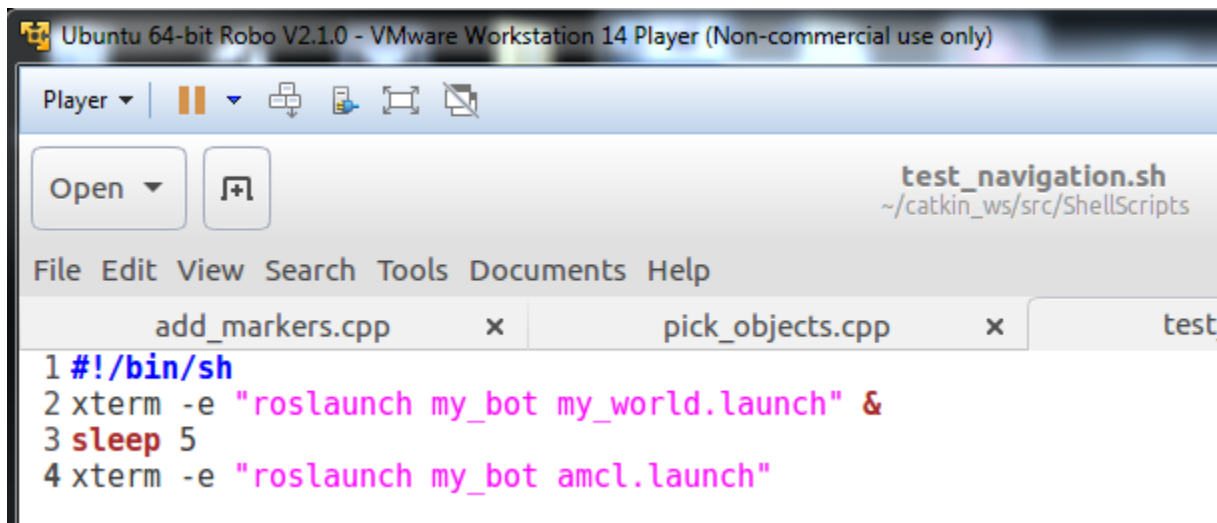
The CMakeLists file also needed editing in this case, specifically in adding compile options for C++11, including catkin directories, adding executability to the node, and specifying linkage to target link libraries.

After giving the robot adequate time to fully map the environment, the “map_server” package’s map_saver function was implemented, allowing the generated map to be saved in .pgm and .yaml files. The .pgm file looked as such:



Navigation:

In order to test navigation, the AMCL package created for the localization project (earlier in term 2) was implemented. In testing the localization and navigation of the AMCL node, a few parameters were altered. The robot had the tendency to run into walls frequently, so the obstacle inflation was raised to 0.5. Additionally, within the base local planner, the “pdist”, “gdist”, and “occdist” scales were altered to place greater weight on avoiding obstacles and following closely to its provided path. The shell script appeared as such:



The screenshot shows a terminal window titled "Ubuntu 64-bit Robo V2.1.0 - VMware Workstation 14 Player (Non-commercial use only)". The window has a menu bar with "File", "Edit", "View", "Search", "Tools", "Documents", and "Help". Below the menu bar, there are three tabs: "add_markers.cpp", "pick_objects.cpp", and "test". The "test" tab is active, showing a shell script with the following content:

```
1 #!/bin/sh
2 xterm -e "roslaunch my_bot my_world.launch" &
3 sleep 5
4 xterm -e "roslaunch my_bot amcl.launch"
```

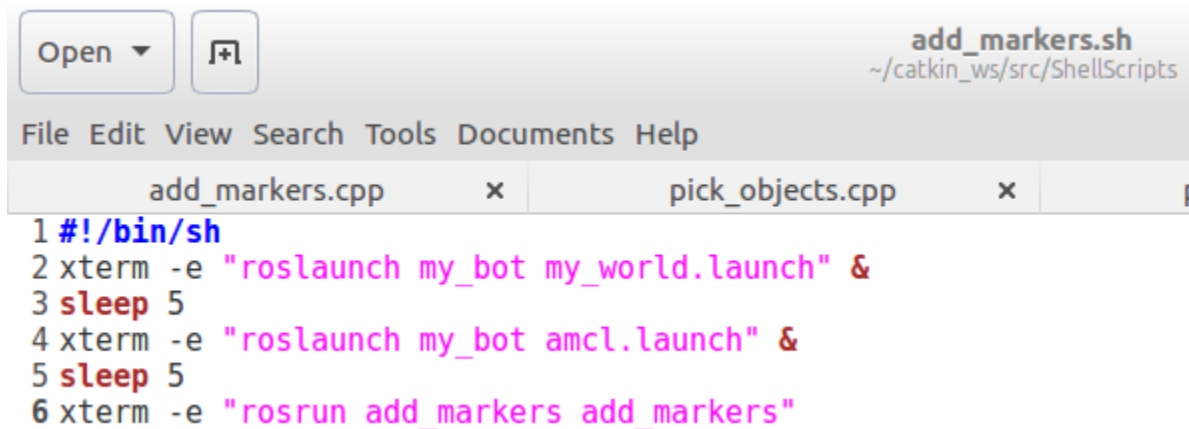
The next step in navigation involved the construction of a “pick_objects” node. This node followed the example set by the template provided in previous lessons, expanding to include two goal locations. The node name was changed to “pick_objects”, and the frame_id was changed to “map”. Then, ROS was instructed to sleep for 5.0 seconds after reaching the pick-up zone, followed by providing the second goal (located towards the tip of the map’s right wing). When reaching each successful goal, a message was coded to appear in the node’s terminal. The shell script for this task appeared identical to that above,

except with “roslaunch my_bot my_world.launch” added below. CMakeLists was edited as well, in the manner described previously (for the C++ node).

Markers:

In order to allow for the publishing of markers as desired by the project guidelines, a few changes needed to be made to the provided tutorial code. This tutorial code was done with the intention of publishing a different object every second. In order to remove this component, the `ros::Rate` variable “r” was commented-out, as well as the “while” loop within which the code was nested in. Therefore, the block of code provided would now only run through once. Additionally, the initial cube was set to a size of 1 meter sides, and this was scaled down to 20 cm sides. The color and orientation were left unchanged.

Next, the lifetime of the first marker was set to a duration of 5.0 seconds, removing it from the map thereafter. Following the publication of the marker, ROS was instructed to sleep for 5.0 seconds and then publish the second marker at a different location for an indefinite period of time. The shell script appeared as such:



The screenshot shows a terminal window titled "add_markers.sh" with the path "~/catkin_ws/src/ShellScripts". The window has a menu bar with "File", "Edit", "View", "Search", "Tools", "Documents", and "Help". Below the menu bar, there are two tabs: "add_markers.cpp" and "pick_objects.cpp". The terminal content is as follows:

```
1 #!/bin/sh
2 xterm -e "roslaunch my_bot my_world.launch" &
3 sleep 5
4 xterm -e "roslaunch my_bot amcl.launch" &
5 sleep 5
6 xterm -e "roslaunch add_markers add_markers"
```

Home Service Robot:

In order to accomplish the listed project tasks, a few modifications needed to be made to both the `pick_objects` and `add_markers` nodes. With regard to the `pick_objects` node, a publisher was created, called “`cube_pub`”, instantiated under the topic `/checkpoints` to publish string messages. While initially trying to simply publish strings directly, this proved to have functionality issues. As such, after much researching, it was found that creating the message variable as a pointer and transferring the intended message as the data for said pointer was properly functional. Therefore, after reaching both pick-up and goal zones, a new string pointer was created and input with a different value, to be published via “`cube_pub`” directly thereafter.

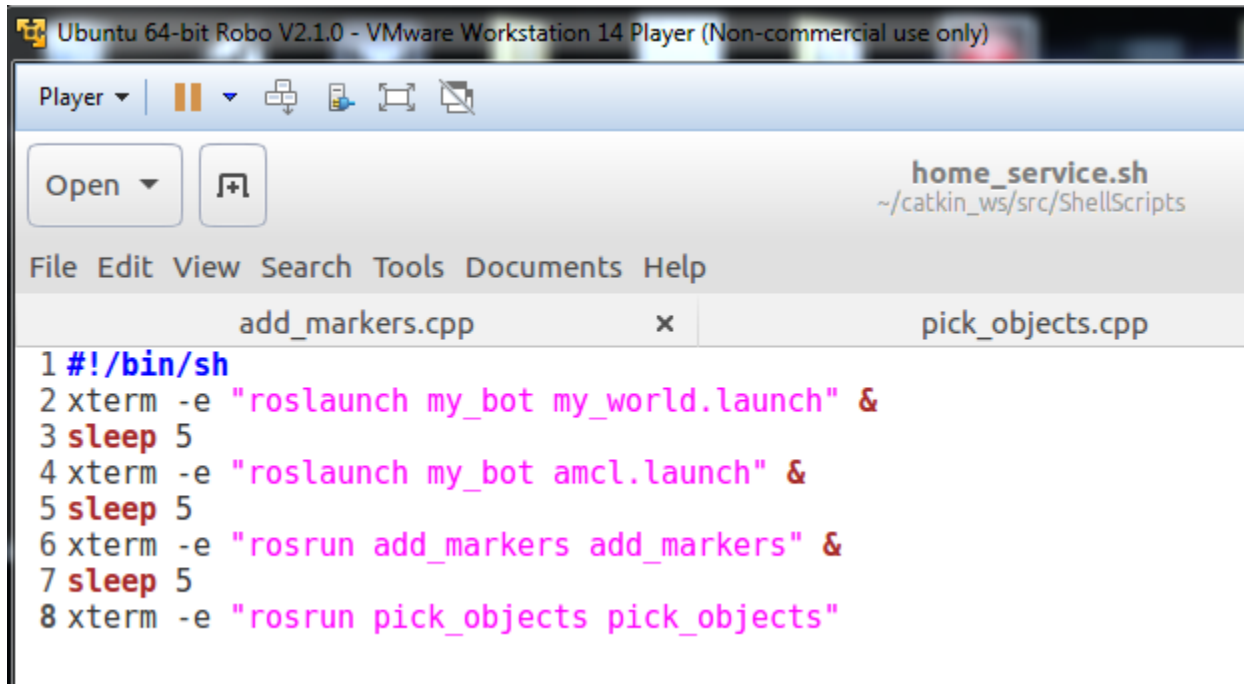
With respect to the `add_markers` node, a few different methods were implemented to try and interact with the `pick_objects` publisher. A subscriber was created, with an appropriate callback function (called “`goal`”) also coded. Much trouble was once again had with being able to directly use string variables, and attempts were made to use constant pointer values instead. After using the “`std_msgs::StringConstPtr`” variable type for the callback function (as detailed in ROS’ tutorial for subscribers), data published on the `/checkpoints` topic was properly relayed to the `add_markers` node. The initial thinking of using the callback function was to implement a “`counter`” variable, changing it based on the incoming subscriber data. However, trouble was met due to the fact that the main function no longer ran as a loop, not allowing the updated counter variable to be noticed.

Therefore, a simpler approach was taken in reading data from the `/checkpoints` topic. After publishing the first marker, the main function was

instructed to simply wait for a message from the /checkpoints topic. If the message of successfully reaching the pick-up location was received, the marker was hidden. Doing so was initially attempted via just using the “marker.action” DELETE functionality, although unfortunately it appears that this component is non-functional. Therefore, a new marker was created (with the same name and ID as the first), but simply with a scale of zero. This effectively removed the first marker.

Afterwards, the main function was again instructed to wait for a message from the /checkpoints topic. Upon receiving information that the goal was reached, yet another marker was created (of identical size and color as the first) at the goal location.

The resulting shell script was as such:



```
1 #!/bin/sh
2 xterm -e "roslaunch my_bot my_world.launch" &
3 sleep 5
4 xterm -e "roslaunch my_bot amcl.launch" &
5 sleep 5
6 xterm -e "roslaunch add_markers add_markers" &
7 sleep 5
8 xterm -e "roslaunch pick_objects pick_objects"
```