Map My World Project

Douglas Teeple

Abstract—Simultaneous Localization and Mapping (SLAM) is an algorithm that solves the problem of generating a map in an unknown world and also performing Localization (determination of the location of the robot) simultaneously. A project implementing FastGraph SLAM in the ROS (Robot Operating System) environment is described and the results are discussed.

Index Terms—Robot, IEEEtran, Mobile Robotics, Kalman Filters, Particle Filters, Localization, Mapping, SLAM, graphSLAM.

1 Introduction

SLAM in robotics comprises Localization - determining a good approximation of the current position of a robot given uncertainties due to noisy sensors and imperfect actuators moving the robot while simultaneously Mapping the unknown environment with no prior given map.

This project consists of creating a ROS (Robot Operating System) catkin workspace that includes a launch script, launch files for gazebo and RVIZ, a robot definition including integration of a Kinect RGBD camera with an associated depth image to laser scanner node, teleop and interactive markers to move the robot in the environment and RTAB-Map [1], [2] (an implementation of fast GraphSLAM) [3]. The project software generates maps and shows loop detection in two worlds. The first world is the kitchen dining world and a second created using gazebo.

Figure 1 shows the Kitchen Dining world in gazebo.

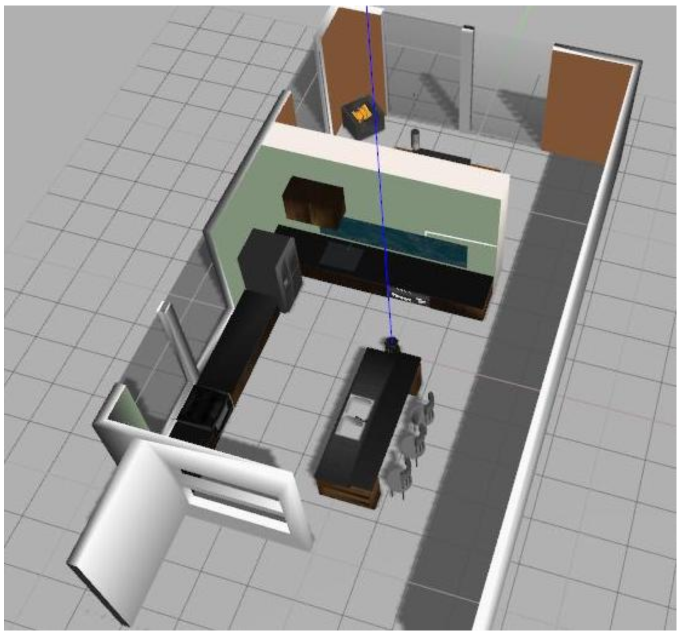

Figure 1. Kitchen Dining World.

2 Background

There are two relevant techniques that may be considered in solving the simultaneous localization and mapping problem. The first is Extended Kalman Filters (EKF) and the second Adaptive Monte Carlo Simulation (AMCL).

2.0.1 *Kalman Filters*

The Kalman Filter is prevalent in control systems. The algorithm is good at taking in noisy measurements in real time and providing accurate predictions of actual variables such as position. However, the Kalman Filter algorithm is based on two assumptions:

- Motion and measurement models are linear.
- State space can be represented by a unimodal Gaussian distribution.

These assumptions limit the applicability of the Kalman Filter as most real robot scenarios do not meet these assumptions.

The Extended Kalman filter (EKF) addresses these limiting assumptions by linearizing a nonlinear motion or measurement function with multiple dimensions using multi-dimensional Taylor series.

While the EKF algorithm addresses the limitations of the Kalman Filter, the mathematics is relatively complex and the computation uses considerable CPU resources.

2.0.2 *Particle Filters*

Particle Filters operate by uniformly distributing particles throughout a map and then removing those particles that least likely represent the current position of the robot.

The Monte Carlo Filter is a particle filter that uses Monte Carlo simulation on an even distribution of particles to determine the most likely position. Computationally it is much more efficient than the Kalman Filter. Also Monte Carlo localization is not subject to the limiting assumptions of Kalman Filters as outlined above.

2.0.3 *SLAM*

Neither of these approaches are adequate to solve the SLAM problem. One approach to solving SLAM is to add the robot pose weight and map to each particle. This approach is problematic because the map is modeled with many variables resulting in high dimensionality. The straight particle filter approach to SLAM in this form scales exponentially and is impractical.

SLAM has both continuous and discrete elements. The continuous component of SLAM relates to the fact that a robot continuously collects odometry to estimate the robot poses and continuously senses the environment to estimate the location of objects or landmarks. Both robot poses and object location are continuous aspects of the SLAM problem. The continuous parameter space composed of the robot poses and the location of the objects is highly dimensional. While mapping the environment and performing localization, the robot will encounter many objects and have to keep track of each one of them. The number of variables increases with time making the problem highly dimensional and making it challenging to compute.

The discrete component relates to sensing the environment to estimate the location of objects. The SLAM algorithm must identify if a relation exists between any newly detected objects and previously detected ones in order to understand if it has been in this same location before and should not be remapped. The discrete relation between objects is known as *correspondence*. The discrete parameter space is composed of the correspondence values. It is also highly dimensional due to the large number of correspondence variables. The correspondence values increase exponentially over time since the robot will keep sensing the environment and relating the newly detected objects to the previously detected ones.

Because of the high dimensionality it is infeasible to compute the posterior under unknown correspondence. SLAM algorithms must rely on *approximation* to conserve CPU resources.

The **FastSLAM** algorithm uses a particle filter. Each particle holds a guess of the robot trajectory so the SLAM problem is reduced to mapping with known poses. It solves the SLAM problem with known correspondences by breaking the problem into two phases:

1) **Estimating the Trajectory**: FastSLAM estimates a posterior over the trajectory using a particle filter.
2) **Estimating the Map**: FastSLAM uses a low dimensional EKF (Extended Kalman Filter) to solve independent features of the map which are modeled with local Gaussian.

The two-phase approach of representing the posterior probability with a particle filter and a Gaussian is called the **Rao-Blackwellized** [4] particle filter approach.

Early versions of FastSLAM had the disadvantage in that known landmark positions are assumed, so it is not possible to model arbitrary environments.

The **Grid-based FastSLAM** grid mapping algorithm models the environment using grid maps without predefining any landmark positions. It extends the FastSLAM algorithm to occupancy grid maps, to solve the SLAM problem in an arbitrary environment.

Grid-based FastSLAM each particle holds a guess of the robot trajectory as in FastSLAM. In addition each particle has its own map. The grid-based FastSLAM algorithm updates each particle by solving the mapping with known poses problem using an occupancy grid mapping algorithm.

The following steps are used to adapt FastSLAM to grid mapping:

1) **Sampling Motion** estimates the current pose given the k^{-1} particle previous pose and the current controls u:
$$p(x_t \mid x_{t-1}^{[k]}, u_t)$$
2) **Map Estimation** estimates the current map given the current measurements, the current k^{th} particle pose, and the k^{-1} particle map:
$$p(m_t \mid z_t, x_t^{[k]}, m_{t-1}^{[k]})$$
3) **Importance Weight** estimates the current likelihood of the measurement given the current k^{th} particle pose and the current k^{th} particle map:
$$p(z_t \mid x_t^{[k]}, m^{[k]})$$

GraphSLAM represents the SLAM problem with graphs composed of:

1) Poses.
2) Features from the environment (landmarks).
3) Motion constraints which tie together two poses.
4) Measurement constraints which tie together a feature and a pose.

GraphSLAM performs graph optimization minimizing the error in all the constraints in the graph using a principle known as Maximum Likelihood Estimation (MLE). Likelihood is complementary to probability. Probability estimates the outcome given the parameters while likelihood estimates the parameters that best explain the outcome. In the case of SLAM likelihood estimates the most likely state and feature locations given the motion and measurement observations.

MLE can be solved analytically, but is computationally very expensive. Though not as accurate as the analytical approach solving MLE numerically using gradient descent is much more computationally feasible. It is this algorithm that is used to tackle SLAM.

GraphSLAM can be viewed as a 5 step process:

1) Construct a graph.
2) Define the constraints.
3) Optimize to solve the system of equations.
4) Linearize as most motion and measurement constraints are non-linear.
5) Iterate.

Graph-SLAM complexity is linear with the number of nodes, which increases with the size of the map.

Loop closure identifies when areas of the environment are revisited when mapping. It identifies that images and locations have already been visited instead of identifying them as new locations. Without loop closure the map is not an accurate representation of the environment. With loop closure the map is significantly smoother and is an accurate representation of the environment.

RTAB-Map is used in this project to find loop closures. When a loop closure hypothesis is accepted a new constraint is added to the graph. A graph optimizer then minimizes the errors in the map. When a loop closure is detected errors introduced by the odometry are propagated to all links thus correcting the map.

RTAB-Map uses only odometry constraints, not landmarks to optimize loop closure constraints.

3 Model Configuration

This section describes the structure and contents of the configuration files used in the project.

3.1 World Models

Two world models were used in this project. The first is the Kitchen-Dining model available in gazebo, and the second was created in gazebo for the project.

Figure 2 shows the Kitchen Dining world:

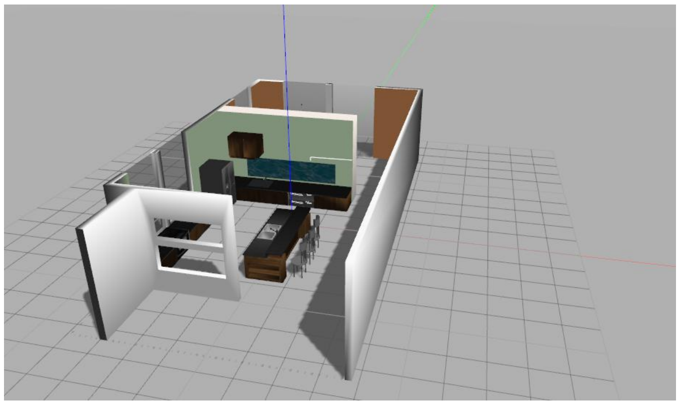

Figure 2. Kitchen Dining World in Gazebo.

3.2 World Creation

The world model *"dougworld"* was created using the gazebo model building facilities.

Figure 3 shows the dougworld model:

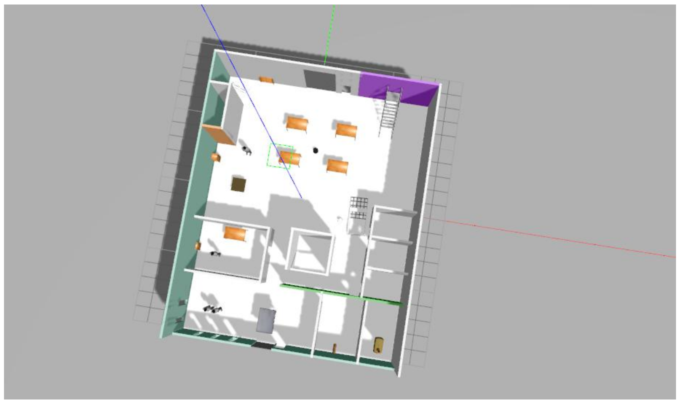

Figure 3. Doug World.

The world file *"doug.world"* was created in gazebo by these steps:

1) Open Gazebo
2) Go to Insert -> Model Database: A layout "OSRF Elevator" was selected from the online model database and then populated with various items such as desks, ladders standing people and tables.
3) Export the model: File -> Save World As, and saved the file as *doug.world* in the worlds directory.

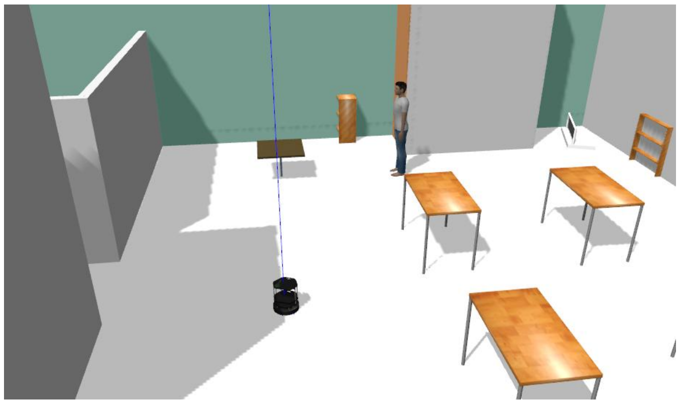

Figure 4. Doug World Closeup Showing Figures and Tables.

3.3 Robot Model

The Robot chosen 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 from course materials as given here:

```
<?xml version="1.0"?>
<robot name="slam_project" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:macro name="turtlebot_sim_3dsensor">
  <!-- RGBD Camera -->
  <gazebo reference="camera_link">
    <sensor type="depth" name="camera1">
        <always_on>1</always_on>
        <update_rate>20.0</update_rate>
        <visualize>true</visualize>
        <camera>
           <pose>0 0 0 pi 0 0</pose>
           <horizontal_fov>1.047</horizontal_fov>
            <image>
                <width>640</width>
                <height>480</height>
                <format>R8G8B8</format>
            </image>
            <depth_camera>
<!-- not used... -->
            </depth_camera>
            <clip>
                <near>0.1</near>
                <far>20</far>
            </clip>
        </camera>
        <plugin name="camera_controller"\\
        filename="libgazebo_ros_openni_kinect.so">
<alwaysOn>true</alwaysOn>
<updateRate>10.0</updateRate>
<cameraName>camera</cameraName>
<frameName>camera_rgbd_frame</frameName>
<imageTopicName>rgb/image_raw</imageTopicName>
<depthImageTopicName>depth/image_raw</depthImageTopicName>
<pointCloudTopicName>depth/points</pointCloudTopicName>
<cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
<depthImageCameraInfoTopicName>depth/camera_info\\
        </depthImageCameraInfoTopicName>
<pointCloudCutoff>0.4</pointCloudCutoff>
<hackBaseline>0.07</hackBaseline>
<distortionK1>0.0</distortionK1>
<distortionK2>0.0</distortionK2>
<distortionK3>0.0</distortionK3>
<distortionT1>0.0</distortionT1>
<distortionT2>0.0</distortionT2>
<CxPrime>0.0</CxPrime>
<Cx>0.0</Cx>
<Cy>0.0</Cy>
<focalLength>0.0</focalLength>
</plugin>
    </sensor>
  </gazebo>
  </xacro:macro>
</robot>
```

3.4 Launch Files

Four launch scripts were created:

1) **mapping.launch**: launches the RTAB-Map map server.
2) **rviz.launch**: launches RViz with the custom *robot_slam.rviz* launch configuration provided as Student Materials.
3) **slam_project_world.launch**: includes the turtlebot robot_description launch file, the world file to use, the Depth-Image-To-Laser-Scan Nodelet, interactive markers and spawns the robot in gazebo.
4) **teleop.launch**: launches the teleop service for robot movement, as provided in Student Materials.
5) **mapping.launch**: launches RTAB-map (Real-Time Appearance-Based Mapping) to produce the map and find closures.

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
  <!-- Arguments for launch file with defaults provided -->
  <arg name="database_path" default="rtabmap.db"/>
  <arg name="rgb_topic" default="/camera/rgb/image_raw"/>
  <arg name="depth_topic" default="/camera/depth/image_raw"/>
  <arg name="camera_info_topic" default="/camera/rgb/camera_info"/>
  <!-- Mapping Node -->
  <group ns="rtabmap">
<node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" \\
    output="screen" args="--delete_db_on_start">
<!-- Basic RTAB-Map Parameters -->
<param name="database_path"       type="string" \\
        value="$(arg database_path)"/>
<param name="frame_id"            type="string" \\
        value="base_footprint"/>
<param name="odom_frame_id"       type="string" value="odom"/>
<param name="subscribe_depth"     type="bool"   value="true"/>
<param name="subscribe_scan"      type="bool"   value="true"/>
<!-- RTAB-Map Inputs -->
<remap from="scan"            to="/scan"/>
<remap from="rgb/image"       to="$(arg rgb_topic)"/>
<remap from="depth/image"     to="$(arg depth_topic)"/>
<remap from="rgb/camera_info" to="$(arg camera_info_topic)"/>
<!-- RTAB-Map Output -->
<remap from="grid_map" to="/map"/>
<!-- Rate (Hz) at which new nodes are added to map -->
<param name="Rtabmap/DetectionRate" type="string" value="1"/>
<!-- 2D SLAM -->
<param name="Reg/Force3DoF" type="string" value="true"/>
```

```
<!-- Loop Closure Detection -->
<!-- 0=SURF 1=SIFT 2=ORB 3=FAST/FREAK 4=FAST/BRIEF \\
        5=GFTT/FREAK 6=GFTT/BRIEF 7=BRISK 8=GFTT/ORB 9=KAZE-->
<param name="Kp/DetectorStrategy" type="string" value="0"/>
<!-- Maximum visual words per image (bag-of-words) -->
<param name="Kp/MaxFeatures" type="string" value="400"/>
<!-- Used to extract more or less SURF features -->
<param name="SURF/HessianThreshold" type="string" value="100"/>
<!-- Loop Closure Constraint -->
<!-- 0=Visual, 1=ICP (1 requires scan)-->
<param name="Reg/Strategy" type="string" value="0"/>
<!-- Minimum visual inliers to accept loop closure -->
<!-- changed from 15 to 5, DT March 8 2018 -->
<param name="Vis/MinInliers" type="string" value="5"/>
<!-- Set to false to avoid saving data when robot is not moving -->
<param name="Mem/NotLinkedNodesKept" type="string" value="false"/>
    </node>
    <!-- visualization with rtabmapviz -->
    <node pkg="rtabmap_ros" type="rtabmapviz" name="rtabmapviz" \\
    args="-d $(find rtabmap_ros)/launch/config/rgbd_gui.ini"\\
    output="screen">
      <param name="subscribe_depth" type="bool" value="true"/>
      <param name="subscribe_scan" type="bool" value="true"/>
      <param name="frame_id" type="string" value="base_footprint"/>
      <remap from="rgb/image"       to="$(arg rgb_topic)"/>
      <remap from="depth/image"     to="$(arg depth_topic)"/>
      <remap from="rgb/camera_info" to="$(arg camera_info_topic)"/>
      <remap from="scan"            to="/scan"/>
    </node>
  </group>
</launch>
```

rviz.launch:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- For visualizing your robots map building in RVIZ -->
<launch>
  <node name="rviz" pkg="rviz" type="rviz" args="-d \\
  $(find slam_project)/launch/config/robot_slam.rviz"/>
</launch>
```

slam_project_world.launch:

```
<launch>
  <arg name="gui"        default="true"/>
  <arg name="world_file" default="$(env SLAM_PROJECT_WORLD_FILE)"/>
  <arg name="base"       value="$(optenv TURTLEBOT_BASE kobuki)"/>
  <arg name="battery"    value="$(optenv TURTLEBOT_BATTERY \\
  /proc/acpi/battery/BAT0)"/>
  <arg name="stacks"     value="$(optenv TURTLEBOT_STACKS hexagons)"/>
  <arg name="3d_sensor"  value="$(optenv TURTLEBOT_3D_SENSOR kinect)"/>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="use_sim_time" value="true"/>
    <arg name="debug" value="false"/>
    <arg name="gui" value="$(arg gui)" />
    <arg name="world_name" value="$(arg world_file)"/>
  </include>
  <include file="$(find slam_project)/launch/includes/\\
  $(arg base).launch.xml">
    <arg name="base" value="$(arg base)"/>
    <arg name="stacks" value="$(arg stacks)"/>
    <arg name="3d_sensor" value="$(arg 3d_sensor)"/>
  </include>
  <node pkg="robot_state_publisher" type="robot_state_publisher"\\
  name="robot_state_publisher">
    <param name="publish_frequency" type="double" value="30.0" />
  </node>
  <!-- Fake laser -->
  <node pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager"\\
  args="manager"/>
  <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"\\
        args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet\\
        laserscan_nodelet_manager">
    <param name="scan_height" value="10"/>
    <param name="output_frame_id" value="/camera_depth_frame"/>
    <param name="range_min" value="0.45"/>
    <remap from="image" to="/camera/depth/image_raw"/>
    <remap from="scan" to="/scan"/>
  </node>
</launch>
```

teleop.launch:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- move that bot -->
<launch>
  <node name="teleop" pkg="slam_project" type="teleop"/>
</launch>
```

3.5 Launch Script

For convenience a Linux bash script starts the components of the simulation:

Listing 1. bot.sh script

```
#!/bin/bash
#################################################################
#
# Udacity Term 2 SLAM Project launch script
#           Douglas Teeple March 2018
#
#################################################################
RED='\e[0;31m'
GREEN='\e[0;32m'
BLUE='\e[0;34m'
YELLOW='\e[0;33m'
CYAN='\e[0;36m'
MAGENTA='\e[0;35m'
BRIGHTRED='\e[1;31m'
BRIGHTGREEN='\e[1;32m'
BRIGHTBLUE='\e[1;34m'
BRIGHTYELLOW='\e[1;93m'
BRIGHTCYAN='\e[1;96m'
BRIGHTMAGENTA='\e[1;95m'
NC='\e[0m' # No Color
function launch() {
        if [[ "$2" != "" ]]
        then
                sleep $2
        fi
        echo -e ${BLUE}"$1"${NC}
        xterm -e $1 &
}
function help() {
        echo -e ${GREEN}"$(basename␣$0):␣[-k]␣[-w<worldfile>]␣[-s<mapfile>]␣\\
␣␣␣␣[--rviz]␣[--gazebo]␣[--localization]␣[--killall]"${NC};
        echo -e ${GREEN}"launch␣SLAM␣Project,␣-k␣keeps␣the␣old␣map,␣\\
␣␣␣␣-w␣names␣a␣world␣file␣-s␣saves␣a␣mapfile"${NC};
        echo -e ${GREEN}"--rviz␣launches␣rviz␣only"${NC};
        echo -e ${GREEN}"--gazebo␣launches␣gazebo␣only"${NC};
        echo -e ${GREEN}"--localization␣performs␣localization␣only"${NC};
        echo -e ${GREEN}"--killall␣kills␣a␣previous␣run"${NC};
}
echo -e ${BLUE}"Starting␣Slam␣Project"${NC}
source devel/setup.bash
worlds=$HOME/catkin_ws/src/slam_project/worlds
world=kitchen_dining.world
keep=0
RVIZONLY=0
GAZEBOONLY=0
LOCALIZATIONONLY=0
KILLALL=0
mapfile=""
for arg in $*
do
        case $arg in
        —h|——help)
                help;
                exit;;
        —k|——keep)
                keep=1;;
        —s*)
                mapfile=${arg#—s};
                keep=1;;
        ——rviz)
                RVIZONLY=1;
                keep=1;;
        ——gazebo)
                GAZEBOONLY=1;;
        ——loc*)
                LOCALIZATIONONLY=1;;
        ——kill*)
                KILLALL=1;
                keep=1;;
        —w*)
                if [ —e ${worlds}/${arg#—w} ]
                then
                        world=${arg#—w};
                else
                        echo —e ${YELLOW}"World ${arg#—w} can't be found, \\
            defaulting to ${world}"${NC};
                fi;;
        *)
                echo —e ${RED}"Invalid parameter: $arg"${NC};
                help;
                exit;;
        esac
done
if (( keep == 0 ))
then
        echo —e ${BLUE}"Deleting map in ~/.ros/rtabmap.db"${NC};
        rm —f ~/.ros/rtabmap.db;
fi
if (( RVIZONLY == 1 ))
then
        launch 'roslaunch slam_project rviz.launch'
elif (( GAZEBOONLY == 1 ))
then
        launch "roslaunch slam_project slam_project_world.launch \\
    world_file:=${worlds}/${world}"
elif (( KILLALL == 1 ))
then
        killall xterm
elif [[ "${mapfile}" != "" ]]
then
        rosrun map_server map_saver —f ${mapfile}
else
        launch "roslaunch slam_project slam_project_world.launch \\
    world_file:=${worlds}/${world}"
        launch 'roslaunch slam_project rviz.launch' 5
        launch 'roslaunch slam_project teleop.launch' 5
        if (( LOCALIZATIONONLY == 1 ))
        then
                launch 'roslaunch slam_project localization.launch \\
        localization:=true' 5
        else
                launch 'roslaunch slam_project mapping.launch \\
        simulation:=true' 5
        fi
        launch "roslaunch turtlebot_interactive_markers \\
    interactive_markers.launch"
fi
```

The script has various options:

1) **-k — –keep**: keep the RTAB-Map database, do not delete it.
2) **-s<mapfile >**: Saves the map file.
3) **-w<worldfile >**: Specifies the world file to use.
4) **–rviz**: launch only RVIZ (useful as RVIZ crashes... a lot.
5) **–gazebo**: launch gazebo only, for generating a world file.
6) **-h — –help**: command help.

3.6 Packages Used

1) **turlebot_viz**: for interactive markers https://github.com/turtlebot/turtlebot_viz.git.
2) **turtlebot**: for the basic turtlebot URDF model https://github.com/turtlebot/turtlebot.
3) **rtabmap_ros**: RTAB-Map https://github.com/introlab/rtabmap.
4) **depthimage_to_laserscan**: Converts Kinect images to laser scans https://github.com/ros-perception/depthimage_to_laserscan.
5) **interactive_markers**: Interactive Marker support https://github.com/ros-visualization/interactive_markers.

3.7 Results

The launch file *"bot.sh"* launches the various components and tools smoothly on Linux. The Jetson TX2 was used as the deployment platform.

3.7.1 Kitchen Dining World

Figure 5 shows the initial pose in RVIZ:

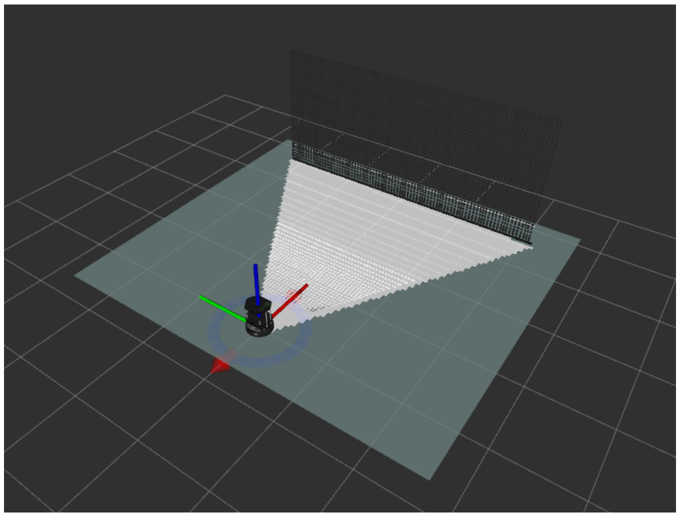

Figure 5. Initial robot pose in RVIZ.

Interactive marker support from turtlebot_viz was added to easily move the robot in the world scene using drag motions https://github.com/turtlebot/turtlebot_viz. The interactive markers are enabled in RVIZ by pressing the *Interactive* button in the top button bar. Dragging the robot around the world accomplishes the mapping.

The turtlebot was outfitted with the Kinect RGBD camera:

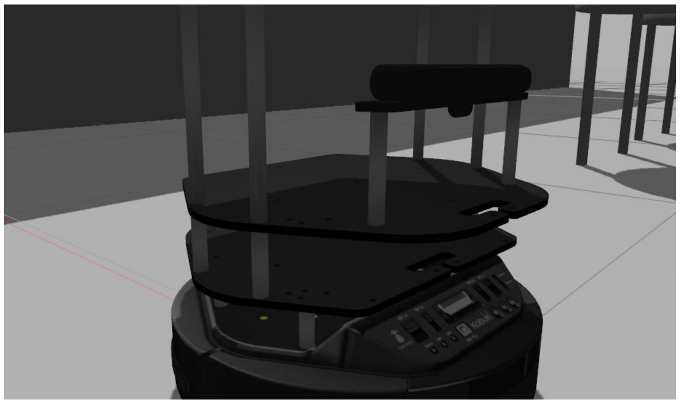

Figure 6. Turtlebot outfitted with Kinect RGBD camera.

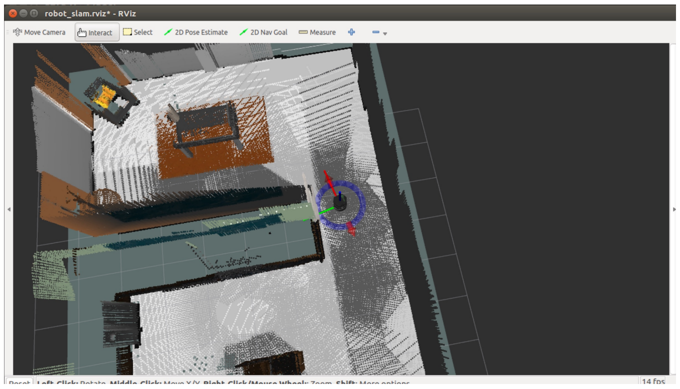

Figure 7. Mapping the Kitchen Dining World.

The mapping service mapped the world as the robot was moved through it. The RTAB-Map mapping module also calculated closures where the robot retraced its path. More than the required 3 closures are recorded in the database *rtabmap.db*. The mapping launch file was modified to be more sensitive to closures. The parameter *"Vis/MinInliers"* was changed from 15 to 5.

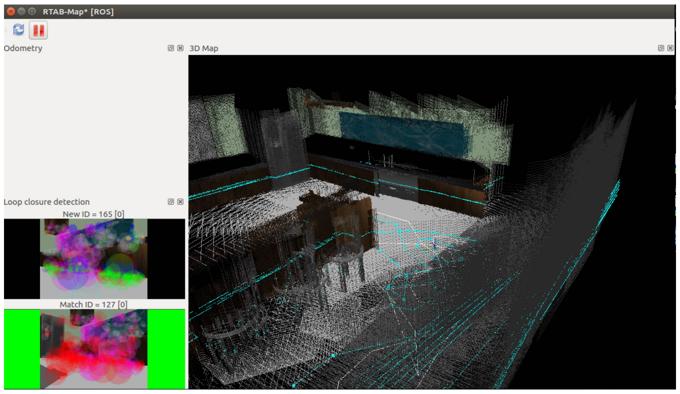

Figure 8. Closures in the Kitchen Dining World in RTAB-Map.

The map database was updated with the Kitchen World map:

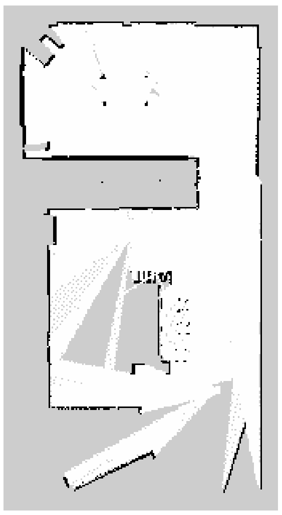

Figure 9. Map of the Kitchen Dining World.

And the corresponding YAML description: **botb.yaml**:

```
image: botb.pgm
resolution: 0.050000
origin: [-4.254636, -8.014119, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

3.7.2 Doug World

The world file *"doug.world"* was created in gazebo. Figure 10 shows the initial pose in *dougworld*:

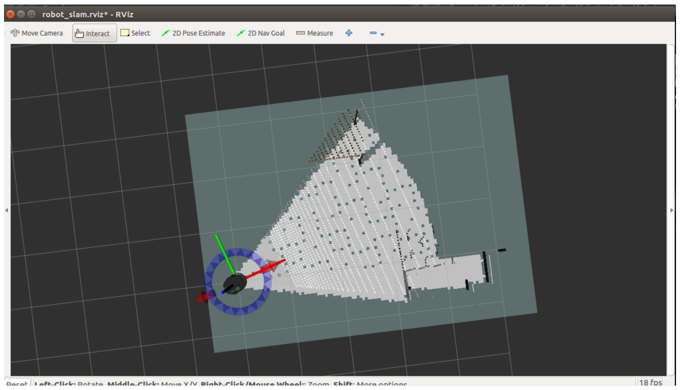

Figure 10. Initial robot pose in RVIZ.

Figure 11 shows the mapping of Dougworld:

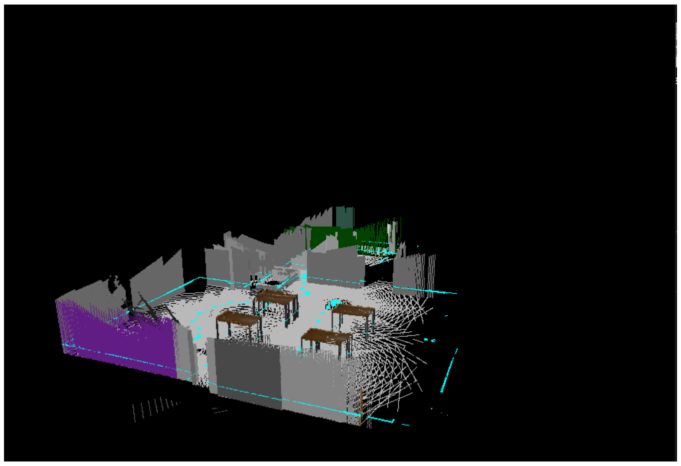

Figure 11. Mapping Doug World in RTAB-Map 3D View.

The mapping service mapped the world as the robot was moved through it. The RTAB-Map mapping module also calculated closures where the robot retraced its path.

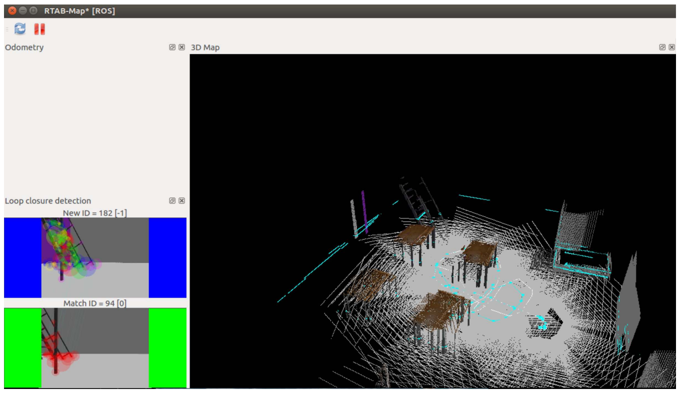

Figure 12. Closures in Doug World in RTAB-Map.

Closures are indicated with a green background in the lower left window.

The map database was updated with the *Doug World* map:

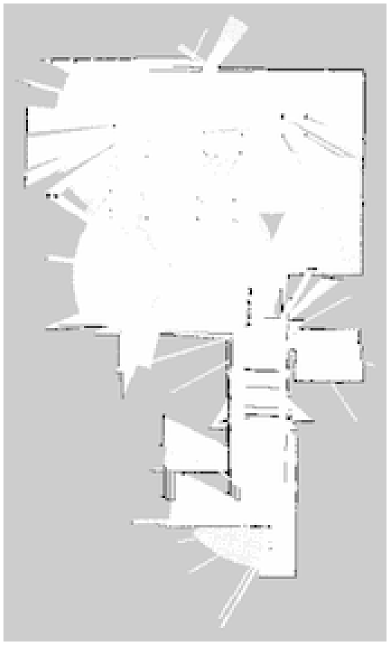

Figure 13. Map of Doug World.

and the corresponding YAML description: **doug.yaml**:

```
image: douga.pgm
resolution: 0.050000
origin: [-6.222296, -14.637760, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

4 Discussion

The project successfully met the course requirements of creating a ROS package that is able to launch your robot and have it map the surrounding area with the models and launch files required to accomplish mapping in two different worlds. 3D maps for both the supplied environment were created. A custom Gazebo world was created and a map for that environment was created as well. At least 3 closures were identified and recorded in the rtab.db database.

While graphSLAM does solve the problem of simultaneous Localization and Mapping, it doesn't work in self-similar environments such as a maze. In the IROS 2014 Kinect Challenge the carpet which had a bold replicated pattern was removed from field of view in order for the algorithm to work. https://github.com/introlab/rtabmap/wiki/IROS-2014-Kinect-Challenge It doesn't handle dynamic environments, but works well for size constrained static environments.

The database created gets large very fast. It can grow to several megabytes in size in just a few moments and does not scale for things very far versus nearby.

GraphSLAM does not solve kidnapped robot problem.

RTAB-Map has many parameters that require tuning for each environment [5] in order for the algorithm to be effective.

5 Conclusion / Future work

The GraphSLAM algorithm successfully performs localization and mapping quickly in the Jetson TX2 environment. SLAM is important for the typical case where a robot is placed in an environment where a map is not given a priori.

The RTAB-Map algorithm performs the important function of detecting closures - areas that the robot has already visited to ensure that mapping points that have already been mapped are not repeated.

The procedure went reasonably well, though the large size of the database often caused the tools to crash, requiring multiple runs to get all the data required by the project. Mapping worked best in environments that were varied - meaning few hallways or rooms that had a similar appearance. The four desks in *Doug World* with a similar appearance made it difficult to get correct closures.

Future work may be in the areas of data size optimization and coping with dynamic worlds.

5.1 Hardware Deployment

1) The project is deployed on a Jetson TX2 board running ROS and Ubuntu 16.04 Linux.
2) Experience shows that this hardware configuration has adequate processing power both in CPU power and memory to host the model.
3) The models were simulated in Gazebo, RViz and RTAB-Map in this project, and no drivers were implemented/integrated to actuate drive motors or read sensors. Kinect hardware and drivers would have to be integrated in order for a hardware version to operate. It would also need GIO connections to drive wheels and be implemented on a suitable platform modeling the simulation robot.
4) A script NVidia speeds up (and resets) the Jetson TX2 clocks. Faster clock speed made the project run much faster. It can be found here: https://github.com/jetsonhacks/installCaffeJTX1/blob/master/jetson_clocks.sh

REFERENCES

[1] M. Labbe and F. Michaud, "Online Global Loop Closure Detection for Large-Scale Multi-Session Graph-Based SLAM," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2661–2666, Sept 2014.

[2] M. Labbe and F. Michaud, "Appearance-Based Loop Closure Detection for Online Large-Scale and Long-Term Operation," *IEEE Transactions on Robotics*, vol. 29, no. 3, pp. 734–745, 2013.

[3] S. Thrun and M. Montemerlo, "The graphslam algorithm with applications to large-scale mapping of urban structures."

[4] M. Labbe and F. Michaud, "Rao-Blackwell theorem,"

[5] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. Leonard, "Past, present, and future of simultaneous localization and mapping: Towards the robust-perception age," *IEEE Transactions on Robotics*, vol. 32, no. 6, p. 13091332, 2016.