3D Mapping of an Gazebo simulation environment using ROS and RTAB-Map

Roberto Zegers Rusche

Abstract—This paper presents a practical guide on how to create a 3D map of an Gazebo environment using RTAB-Map ROS package. First, the main characteristics and categories of SLAM algorithms are briefly described. Then, an insight into the key aspects when configuring RTAB-Map is performed. This includes adding a depth sensor camera into an existing simulated robot and setting up RTAB-Map for that robot. Finally RTAB-Map's 3D mapping capability is tested in a software-in-the-loop configuration on two different Gazebo environments and the mapping results are evaluated.

Index Terms—Robot, RTAB-Map, SLAM, 3D mapping, point cloud, ROS, Gazebo.

1 Introduction

This paper explores the implementation of RTAB-Map, a camera-based Simultaneous Localization and Mapping (SLAM) algorithm. RTAB-Map, which stands for Real-Time Appearance-Based Mapping, relies on a visual depth sensor instead of a traditional laser range scanner for localization and mapping. Through the use of a depth image, RTAB-Map performs graph-based online SLAM with appearance-based loop-closure. This special feature allows RTAB-MAP to create a reconstruction of the environment into a colored 3D model. The experiences reported on this work should provide insight to anyone seeking to implement RTAB-Map as SLAM solution for a mobile robot application.

2 Background

The problem of learning maps is an important problem in mobile robotics. Models of the environment are often inherently necessary for mobile robots to perform their tasks. For instance, path planning requires that the planning algorithm has access to a pre-existing map of the world and its relevant aspects. To build consistent environment maps an accurate and reliable localization source is required, otherwise the captured map data will not be correctly positioned. Estimating a map with known poses can be solved using the occupancy grid mapping algorithm. However when no localization system (e.g. GPS) is available, the localization and mapping problem needs to be solved concurrently. In other words, a map is needed to localize a robot, and a pose estimate is needed to build a map. Simultaneous localization and mapping, known as SLAM, describes the capability of simultaneously constructing a map of the environment and estimating the pose of a robot moving within it [1]. Because SLAM allows autonomous navigation of robots in a previously unknown environment, it is considered a fundamental problem to solve for robots to become truly autonomous. Due to this importance SLAM has been extensively researched in the last two decades and a large variety of different SLAM approaches have been developed. Without going too much into detail, SLAM algorithms can be distinguished along a number of different dimensions; one of them being the probabilistic approach used. According to its probabilistic implementation, a SLAM algorithm will be capable of solving the Full SLAM problem, the Online SLAM problem or both. The Online SLAM problem consists in estimating the current (instantaneous) pose and the map given the current measurements and controls, the previous measurements and controls are not taken in account. The Full SLAM problem consists in estimating the entire path instead of only the instantaneous pose, given all the measurements and controls up to the evaluation time. This characteristic allows accumulated errors in positioning and mapping to be corrected using so called loop-closure events; hence, allowing for long-term accurate performance.

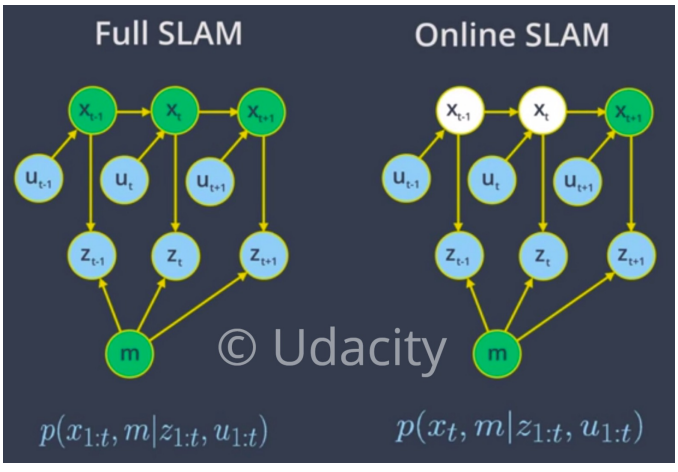

Fig. 1. Graphical Model of Full Slam, Graphical Model of Online SLAM. Source: Udacity

Also a key characteristic of the SLAM problem altogether, is that it possesses both a continuous and a discrete component. The continuous estimation problem pertains to the location of the objects in the map and the robots own pose variables. The discrete nature has to do with correspondence: When an object is detected, a SLAM algorithm must reason about the relation of this object to previously detected objects. This reasoning is typically discrete: Either

the object is the same as a previously detected one, or it is not [2].

The three main families of SLAM algorithms from which most others are derived are the Extended Kalman Filter algorithms, the so called Graph-SLAM algorithms and the Fast-SLAM algorithms. The Extended Kalman Filter algorithms are online SLAM algorithms. The Graph-SLAM algorithms solve the offline Full SLAM problem, that is, they calculate a solution taking into account all measurements taken so far. Therefore Graph-SLAM algorithms are not suitable for real-time and incremental map building. Finally the newest family of algorithms, Fast-SLAM, use a custom particle filter along with a low-dimensional extended Kalman filter to compute the pose and related landmarks in separate particles while maintaining the full path posteriors approach. Fast-SLAM algorithms solve both the Full SLAM problem and the Online SLAM problem.

In this paper we used ROS, as it is a widely adopted robotic platform that provides a large and growing collection of fundamental robotics algorithms already implemented. Popular SLAM approaches available on ROS are [3]:

- **GMapping** and **TinySLAM** are two approaches that use a particle filer to estimate the robot trajectory. GMmapping is a grid-based Fast-SLAM algorithm.
- **Hector SLAM** can create fast 2D occupancy grid maps from a 2D lidar with low computation resources. The approach of this algorithm is to do the so called Fast-SLAM and it can be used without odometry.
- **Karto SLAM**, **Lago SLAM5** and **Google Cartographer** are lidar graph-based SLAM approaches. They can generate 2D occupancy grid from their graph representation. Google Cartographer can be also used as backpack mapping platform as it supports 3D lidars, thus providing a 3D point cloud output.
- **RTAB-Map** and **RGBDSLAMv2**, are graph-based SLAM implementations for RGB-D cameras that can use external odometry as motion estimation, they also generate a 3D occupancy grid and a dense point cloud of the environment.

This paper implements RTAB-Map which uses data from a depth camera (e.g. X-Box Kinect or Intel Real Sense) to localize the robot and map the environment. RAB-Map incorporates a process called loop closure: When the robot revisits an area it has seen before, it compare the current image with the images that have been saved into its database and it will detect if it has been there before. When a loop closure is detected, a new constraint is added to the map's graph, then a graph optimizer minimizes the errors in the map, and accumulated errors in odometry positioning are corrected. This odometry drift correction is published as /map -> /odom TF transform accurately localizing the robot into the map.

3 Robot and RTAB-Map Configuration

The RTAB-Map package assumes that the robot is configured in a particular manner in order to run. As a minimum, RTAB-Map requires a depth camera to function. However the recommended configuration for optimal performance includes a depth camera, odometry data and a 2D laser scanner. In this work the robot model was taken over from a previous project. That robot model includes the laser range finder and the scan topic so they were not taken out and the optimal configuration for RTAB-Map was used.

3.1 Structural and Sensor Upgrades

The robot model used builds on the robot model found in the dumpster ROS Package and bears the name rtab_dumpster. With regards to its structure, the dumpster bin had to be moved some centimeters toward the back in order to make some room in the front for the kinect pole. The laser pole was moved a bit backwards too and it height was incremented so that the laser scanner has a full 360 field of view. Finally a new link called kinect_pole was added and a model of the kinect camera was placed on top. Also a link called camera_rgbd_frame" was added, this is important as the camera requires such a link to be able to configure the orientation of the image.

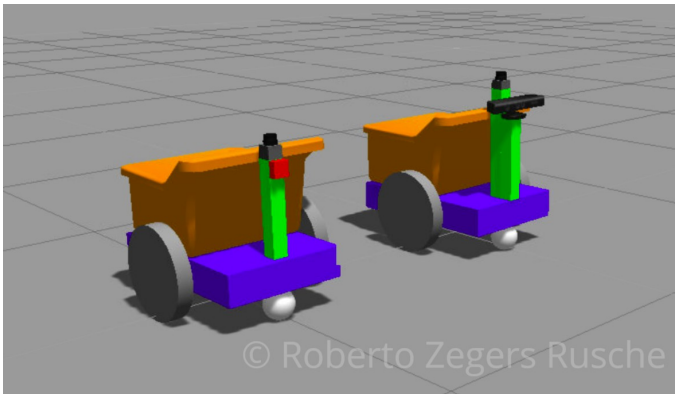

Fig. 2. The original dumpster robot (left) and the new, modified rtab_dumpster model that includes a Kinect depth sensor (right)

Finally a Gazebo plugin has to be added into the Gazebo part of the robot xacro description in order to provide the depth camera functionality in simulation. For this the correct link to the cameras shared object file has to be added, which is "libgazebo_ros_openni_kinect.so". This plugin outputs a raw rgb image from it as a sensor_msgs/Image messages, the camera geometry data in sensor_msgs/CameraInfo message and a depth map message of type sensor_msgs/PointCloud2. Note that this plugin includes a reference tag which has to match the name of the camera link as defined on the robot xacro file (in this particular case the rtab_dumpster.xacro file). Also the sensor name must be unique from all other sensor names. Next, the topic names the camera will be publishing on, have to be set. RTAB-Map defaults to the topics /odom, /scan, /rgb/image, /depth/image and /rgb/camera_info, but those topics can be remapped from the RTAB-Map launch file if necessary. The parameter "frame_id" defines the root frame used by the robot in its URDF tree structure. Depending on the particular name given it could be base_footprint, base_link, robot_footprint, L0 or any other

name. Make sure the ”frame_id” parameter matches the name of the root frame of the robot.

Finally the coordinate frame of the image that is published under the tf tree has to be set.

3.2 RTAB-Map Configuration

RTAB-Map has many parameters that can be configured from its ROS Package launch file. For this project a basic template launch file (mapping.launch) was provided as part of the student materials. To reproduce the work, instead of creating a launch file from scratch it is highly recommended to start by copying a exemplary launch file, such as the one provided on the RTAB-Map ROS Package Github page under rtabmap_ros/launch/rtabmap.launch, and modify it.

In this launch file it is possible to differentiate between two groups of parameters: the RTAB-Map ROS parameters and the RTAB-Map algorithm parameters. The package ROS wiki page (http://wiki.ros.org/rtabmap) is the best place to reference any of the packages parameters. A description of the RTAB-Map algorithm parameters is beyond the scope of this paper. RTAB-Map ROS parameters are used for connecting the RTAB-Map library with ROS. Some of them are for instance frame ID, odom frame, subscribe depth and subscribe scan. It is important to set the subscribed topics right. RTAB-Map subscribes to certain topics in order to get information for instance from the camera or from the laser. The topics that RTAB-Map ROS subscribes to are ”rgb/image”, in order to get a RGB monocular image of what the camera is capturing, it also subscribes to ”rgb/camera_info” in order to get the camera metadata, then it subscribes to ”rgbd_image” which is a topic from where it gets the depth image that the kinect camera is capturing and finally the /scan topic for the laser scan data and the /odom topic for capturing the odometry data. Configuring this right is very important because by default the name of the topics that RTAB-Map package will subscribe are these ones, but the name of the topics are different in each robot. Topic names can be changed at runtime through remapping to the actual names of the topic that are currently running.

Note: The description above only depicts how to quickly get started with a minimal configuration of RTAB-Map.

4 Checking That Everything Works

First execute RTAB-Map by running these two commands and look for error messages on the console:

```
$ rtab_dumpster rtab_dumpster.launch
$ roslaunch rtab_dumpster mapping.launch
```

We can also see the structure of how topics are passed around the system. With both windows running, open a new terminal window and type in:
$ rosrun rqt_graph rqt_graph

This command generates a representation of how the nodes and topics running on the current ROS Master are related. Use this tool to check that all connections are established as intended.

Next, check what the running nodes are publishing by viewing all the topics currently registered. Use the command $rostopic list to get a list of ROS topics from the ROS master. Verify that at minimum following topics are being published:

```
/camera/depth/camera_info
/camera/depth/image_raw
/camera/depth/points
/camera/rgb/camera_info
/camera/rgb/image_raw
/scan
/odom
```

Note: Gazebo in its version 7.0.0 seems to have a bug that prevents the simulated Kinect sensor to publish its topics. This issue can be quite common since ROS Kinetic full desktop version will install Gazebo version 7.0.0 along it by default. Upgrading to Gazebo 7.11 gets the problem solved.

The rostopic command-line tool can also be used to display information about the publishing rate of those topics. Note that it is possible to execute rostopic hz and pass in multiple topics separated by a space, see Figure 3.

Fig. 3. Image shows rostopic hz command run with multiple topics. All required topics are publishing data

Once there is laser data, camera data and odometry data published into the correct topics one can move on to conduct another system check in RVIZ. If the visualizer is not already running, start it with:
$ rosrun rviz rviz

For this project a Rviz display config file (robot_slam.rviz) was provided as part of the student materials. If this file is not available Rviz can be configured as follows:

Add an Image display and set the Image Topic to rgb/image_raw. To visualize a depth image add a new Image display with topic name depth/image_raw.

Finally in the Displays panel, click Add and choose the PointCloud2 display. Set its topic to /camera/depth_registered/points and set Color Transformer to

RGB8. You should see a color, 3D point cloud of the scene in front of the robot. See Figure 4.

The scene displayed by Rviz on startup is a good reference for debugging camera frame rotation issues. It is very important to make sure the 3D point cloud is displayed in correct rotation. If i is not, modify the XACRO file and change the corresponding joint orientation and if necessary the mesh orientation. In this particular case the rgbd_frame (the frame the gazebo plugin is in) had to be rotated.

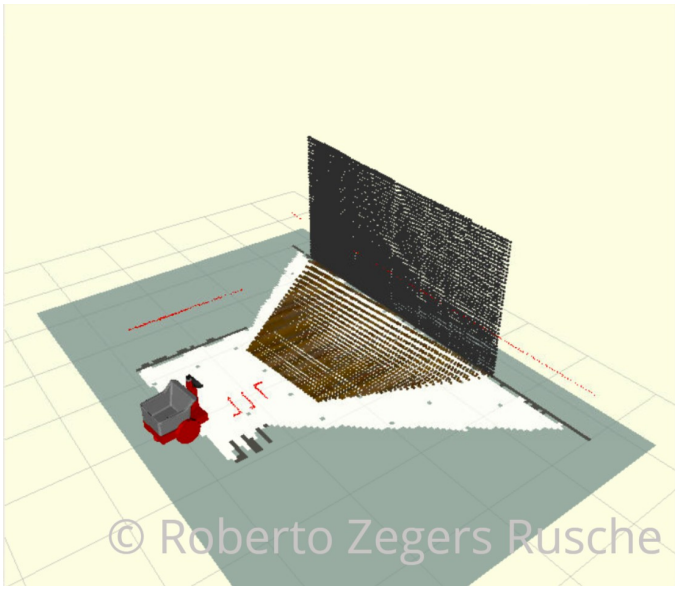

Fig. 4. The scene displayed by Rviz on startup showing a proper configured depth sensor

5 Scene Configuration

Two different Gazebo worlds were used to test the RTAB-Map algorithm. The first Gazebo world, called **kitchen_dining.world**, was provided as part of the classroom material. It represents a tiny urban apartment that features a living area and an open kitchen-dinning island with three barstools. See Figure 5.

Fig. 5. Top view of the supplied kitchen_dining Gazebo world used to evaluate the performance of the RTAB-Map algorithm

The second Gazebo world, called **stockroom.world**, represents a small stockroom as on a typical industrial area and was developed from scratch by the author. See Figure 6.

The stockroom world contains several exclusive SDF models specially designed for this project. Those models were created from meshes obtained from Googles SketchUp 3D Warehouse. Other models were obtained from public Gazebo collections or found in GitHub. The sources of all meshes used for the custom SDF models are properly named in each models config file. See Figure 7.

Fig. 6. Top view of the custom build stockroom Gazebo world with the custom build robot in it. Screen capture from Gazebo.

As world files are XML-based scene description files that encode the layout of all objects present in the environment, all required objects were first inserted into the scene on a arbitrary position using a text editor. Next the scene was adapted in a graphical way using Gazebo tools for moving and rotating objects to their final positions. Because most custom models include a static tag set to true (they are not affected by gravity) it was very important to verify that the vertical axis positioning was right, otherwise they would appear to be floating. It is worth noting the 3d space was filled with many single unrepeated 3D models in order to increase the number of unique visual features. Also, when arranging objects in a scene, care was taken to mimic a realistic stockroom scene.

The code for the **kitchen_dining** Gazebo world is the rtab_dumpster ROS Package and the code for the **stockroom** Gazebo world is the stockroom ROS Package. See section 10 for instructions on how to download and use these ROS Packages for testing the RTAB-Map algorithm in simulation.

6 Mapping Process

Two launch files are required to run the necessary nodes for mapping:

Fig. 7. Some of the custom SDF 3-D Models created for the project

$ roslaunch rtab_dumpster rtab_dumpster.launch
$ roslaunch rtab_dumpster mapping.launch

This will launched Rviz with a predefined configuration that shows the mapping process.

In ROS, many teleoperation packages exist to drive a robot manually. This control can either be through keyboard, joystick, or some other input such as the rqt_robot_steering plugin. In this example the teleop_twist_keyboard package was used. With Gazebo and Rviz running, the teleop package has to be run in a new terminal:

$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py

With the terminal window of teleoperation node selected, the robot can be moved according to the on-screen controls.

In brief, the mapping process followed these steps: the teleoperator sent motion commands and then observed how they affected the robot and the mapping of the environment. A 3D point-cloud map of the environment got built incrementally in Rviz as the robot moved to unexplored spaces. When the operator was satisfied with the area covered the mapping processes was terminated closing RTAB-Map from the terminal window.

Once RTAB-Map was terminated a database was saved. This database was copied or moved to a new folder before moving on to map a new environment. This is because relaunching the mapping node deletes any database in place on launch start up.

RTAB-Map provides two ways to view the databases created.
$ rtabmap ~/.ros/rtabmap.db
$ rtabmap-databaseViewer ~/.ros/rtabmap.db

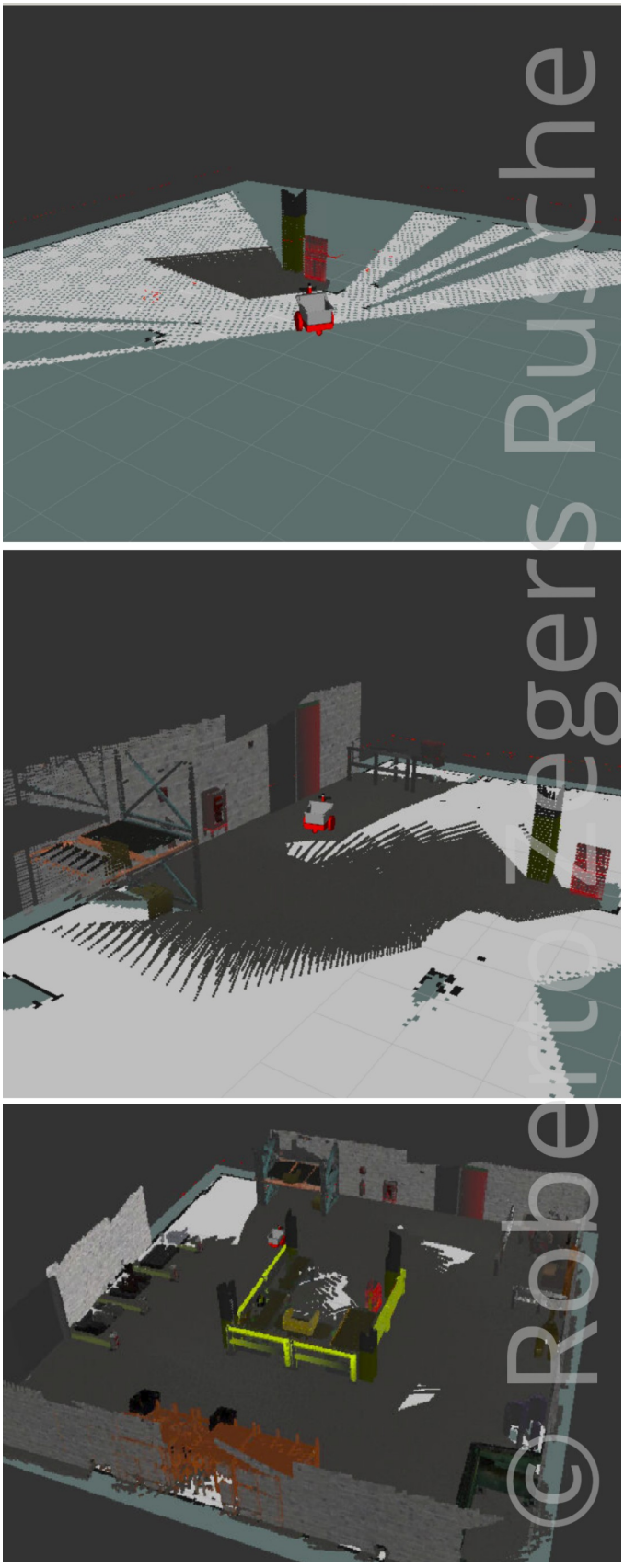

Fig. 8. Image sequence showing the different stages of mapping progress of the custom build stockroom world

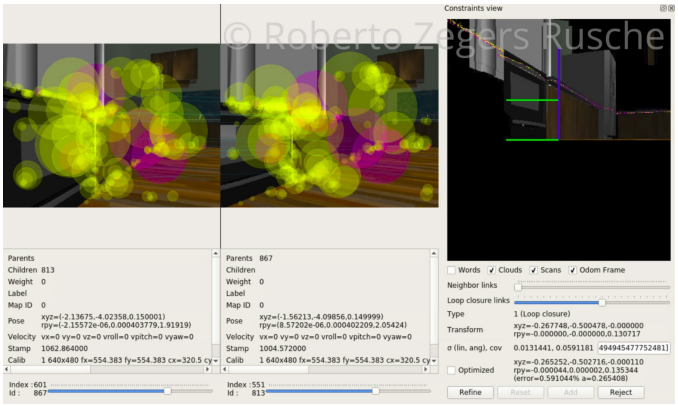

Fig. 9. Screen capture of the rtabmap-databaseViewer tool showing two images in the in the database and the detected loop closures (in pink)

Note that executing RTAB-Map with Gazebo requires sufficient hardware performance to be able to run properly. A server running an single socket dual core Intel Xeon CPU processor @ 2.30GHz (hyperthreading enabled), 16GB of RAM, and a NVIDIA Tesla K80 GPU processor was used in this work as host.

7 Results

This work showed that RTAB-Map can be configured correctly in just a few work hours without prior knowledge. It also demonstrated the capacity of RTAB-Map to successfully create a 3D map of two different simulated environments. Next, images of the mapping process are shown (Figure 8).

Under the configuration used for this project, RTAB-Map was able to find 27 loop closures on the stockroom world.

These results prove that RTAB-Map is able to create a 3D point cloud of a simulated environment.

It was also shown that RTAB-Map can effectively use this information to localize a robot robustly in the map and plan a path and generate the motor commands required for the robot to reach its goal.

Finally, any software-in-the-loop simulation system, such as this one based on Gazebo, will add significant computational load on any system. This work proves that, when Gazebo is not limited to the hardware performance, RTAB-Map is fast enough for realtime mapping even in simulated environments.

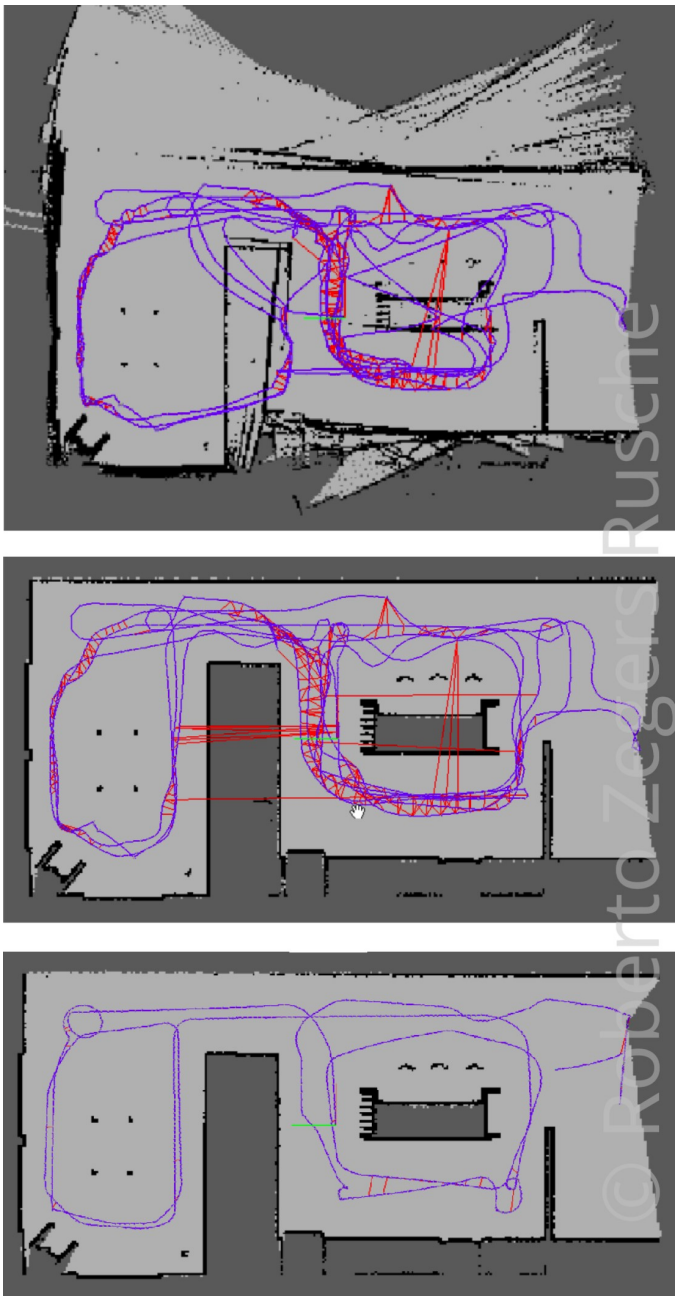

Fig. 10. Different mapping attempts as shown by the Database Viewer, final image represents result with final RTAB-Map configuration. The red edges are loop closure constraints recognized by RTAB-Map.

8 Discussion

Mapping an unknown environment is a problem that has been widely studied for a long time and is still considered to be a challenging task. RTAB-Map is a 3D mapping and localization package that fuses LaserScan messages, odometry and color and depth images from a RGBD-camera to create a database that can be used for SLAM. The algorithm was tested in two simulated worlds and in both cases very satisfactory results were obtained.

One of the lessons learned from mapping with RTAB-Map is that, without prior experience, one may require multiple mapping attempts in order to get satisfying results. The slower the operator moves the robot, avoiding getting stuck and bumping into obstacles, the better the map will be. It is also better to occasionally do a full 360 turn. This will enable the system to see more images at different angles, which can result in more loop closures. This is an important consideration to keep in mind, as RTAB-Map requires a sufficiently structured environment and stable features to obtain reliable results. The drawback of this requirement is that sometimes environments do not present such conditions. Because RTAB-Map will be unable to find loop closures in dark environments and featureless places, those environment conditions will causes the algorithm to fail. This is specially a common problem with simulated environments, that can be visually featureless and have

repetitive textures that can cause wrong loop closures. Erroneous loop closure is a particularly problematic situation because if even one wrong loop closure is incorporated into the map, the algorithm can fail catastrophically, producing a map with increasing deformation. One can reduce the false loop closures by decreasing RGBD/OptimizeMaxError parameter (default 1 meter) to something like 0.1 meter or 0.05 if the robots odometry is good and does not drift too much.

Another insights acquired during the development of this work is that building a custom Gazebo World represents a challenge per se in its realization. In times were it is said that "content is king", meaning that content is central to the success of a platform, Gazebo models are way too scarce. This problem is exacerbated even further considering the complexity of building a custom model. Finding the necessary 3D mesh, making the model's SDF file and testing the results is a huge hassle that involves significant effort. But, even though the models developed for this paper included just the bare minimums (e.g. using the visual mesh as collision element, setting no inertial properties, keeping models reduced to one link), the process of building SDF models was very frustrating because often Gazebo failed to deliver the expected results. Gazebo does not displays any message when encountering errors related to a sdf file or a mesh. Meshes are displayed inconsistently across different graphic library clients, eg. Gzweb. Occasionally, for unknown reasons, meshes display polar rendering artifacts. Sometimes white lines appear on objects, those white lines are not random as they always match the edges of the mesh geometry. At times renders are broken and models are displayed incomplete. And last but not least the cumbersome user interface for moving and rotating both objects and the users viewpoint result in a subpar user experience.

Finally, in the example of RTAB-Map we became aware of the ability of leveraging existing ROS packages for performing complex tasks that otherwise would take months if not years to develop from scratch. This is just plain and simply fantastic. In the current age where managing the ever scarcer time resource is perhaps the greatest challenge of all, being efficient is key to the success of many projects. In this sense reusing what has already being developed by others will result in important time savings in the creation of any robotic solution. This could be one of the most interesting and most important contributions of the ROS framework to the field of robotics.

9 Future work

Future work could aim to replicate our results on a real robot in a real environment. In that scenario new challenges such as systematic odometry error in wheel odometry or camera calibration are expected. A thorough examination of RTAB-Map's parameters should be conducted as a starting point when looking forward to a real world 3D map building project.

Future work could also study the implementation of visual odometry on robots that lack wheel odometry such as Unmanned Aerial Vehicles (UAVs). The ability to offer real time localization without the need of any other sensors is certainly one the most appealing features of RTAB-Map and a very interesting topic for future work.

Finally, future studies could also investigate the use of RTAB-Map for scanning outdoor environments. Outdoor appearance change constantly because of illumination variations, shadow dynamics, changing weather conditions, changes in vegetation across seasons and moving objects. The characteristics of uncontrolled outdoor environments make them a hard case for any visual localization method, a hot topic and an active area of research.

10 Reproduce the work

10.1 Requirements

These instructions assume a standard ROS Kinetic Full Desktop installation on Ubuntu 16.04 (Xenial).

10.2 Installation steps

1) To get started, two git repositories need to be cloned into the src directory of a local catkin workspace:
$ git clone https://github.com/rfzeg/rtab_dumpster.git
$ git clone https://github.com/rfzeg/stockroom.git

Then, install these ROS Packages only if not already present:
$ sudo apt-get install ros-kinetic-teleop-twist-keyboard
$ sudo apt-get install ros-kinetic-rtabmap-ros

2) The packages need to be build using catkin_make:
$ cd ~/catkin_ws
$ catkin_make

And the catkin workspace has to be sourced:
$ source ~/catkin_ws/devel/setup.bash

In order to get Gazebo to find the new models, this line must be added to the system's ~/.bashrc file:

export GAZEBO_MODEL_PATH=~/catkin_ws/src/stockroom/models:$GAZEBO_MODEL_PATH

Finally add model collision modelling to the kitchen_dining room world:

$ curl -L https://s3-us-west-1.amazonaws.com/udacity-robotics/Term+2+Resources/P3+Resources/models.tar.gz
tar zx -C ~/.gazebo/

3) a) To run Gazebo's **kitchen_dining world** with the rtab_dumpster robot ready for mapping, the next commands need to be executed in order:

$ roslaunch rtab_dumpster rtab_dumpster.launch
$ roslaunch rtab_dumpster mapping.launch
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py

3) b) To run Gazebo's **stockroom world** with the rtab_dumpster robot ready for mapping, the next commands need to be executed in order:

$ roslaunch stockroom stockroom.world
$ roslaunch rtab_dumpster mapping.launch
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py

Rviz can then be used to visualize the map building process.

4) To perform localization, instead of mapping launch the 'localization.launch' file inside the rtab_dumpster package.

5) To open the RTAB-Map database viewer type:
$ rtabmap-databaseViewer

REFERENCES

[1] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," vol. 32, no. 6, pp. 1309–1332, 2016.
[2] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT press, 2005.
[3] M. Labb and F. Michaud, "Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446.
[4] "Advanced parameter tuning tutorial." http://wiki.ros.org/rtabmap_ros/Tutorials/Advanced%20Parameter%20Tuning. Accessed: 2019-02-27.
[5] "Ros rtab-map package summary page." http://wiki.ros.org/rtabmap_ros/. Accessed: 2019-02-27.