Robotics project

Imparted by Dr. Alf Kjartan Halvorsen

Simultaneous Localization and Mapping
Final report

Marcos Eduardo Castañeda Guzman
A01372581@itesm.mx

Gerardo Uriel Monroy Vázquez
A01372286@itesm.mx

Emmanuel Hernández Olvera
A01371852@itesm.mx

02/12/19

Table of contents

Introduction

Localization and navigation are the most important tasks of autonomous mobile robots. Simultaneous localization and mapping (SLAM) is the process of creating a map using a robot or unmanned vehicle that navigates that environment while using the map it generates[9]. The main principle of SLAM is to detect the surrounding environment using the information gathered through the sensors on the robot, and to reconstruct the map of the environment and at the same time estimating the pose (including both location and orientation) of the robot. Since SLAM was first put forward in 1988, it was growing very fast, and many different schemes have been formed. The implementation of this project considered the use of a commercial robot from Dr. Robot, the x80. The x80 robot is a two-wheeled locomotion robot with differential drive specifically designed for researchers developing advanced robot applications such as remote monitoring, telepresence and autonomous navigation/patrol. This robot is not sold with a camera. To process all the information coming from the surroundings a PC was added to the system(UP board) and an rgb-d camera Intel real sense SR300 was added too. To develop slam an odometry node is needed, this odometry information can be processed using camera messages or using the encoders of the wheels, in this SLAM implementation the odometry was calculated using the encoders information.

Mechanical subsystem

Mechanical design

The mechanical requirements of the system, detailed in the Design Requirements Document, have been implemented and satisfied taking into account that the project did not require to build a mobile robot from scratch but to use an existing robot provided by the main stakeholder (DrRobot X80); the mechanical design was not proposed. It was only necessary to add one item to complete the requirements. To attach the stereoscopic camera to the chassis, an additional assembly was needed: a mobile base actuated by servomotors already exists on the robot.

In the last report it was stated that the mount to be used was the one for the Intel Realsense R200 camera, but later as the team performed some tests for the map creation the R200 camera did not perform well as the expectations for these elements. Given this circumstance, the team decided to switch to the Intel Realsense SR300 camera (best known as BlasterX Senz3D [1]), which outputed

better results. That is the reason why it was necessary to include a different camera mount. The conditions for the camera mount were simply to have the correct measurements to be attached via four screws onto the metal base and to have enough clearance with the pentagonal top piece so the camera wouldn't collide with it when rotated. The measurements taken on both metal pieces are shown in Fig. 1.

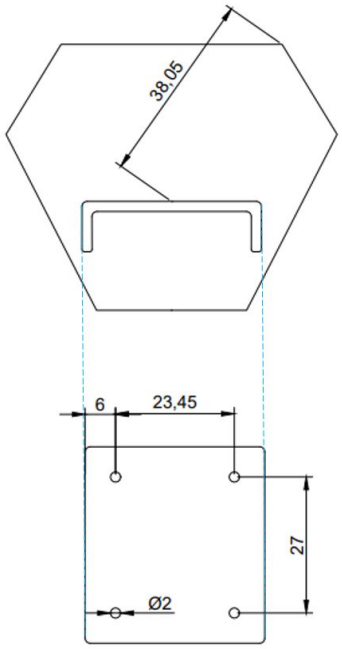

Fig. 1 Measurements to satisfy for the camera mount

Mechanical implementation

As stated before, only one element in the system was designed and implemented: a mount for the SR300 camera. For illustrative purposes the mechanical design of the X80 mobile robot is shown with a couple of its measurements in Fig. 2. This schematic was obtained from the DrRobot X80 Manual [2].

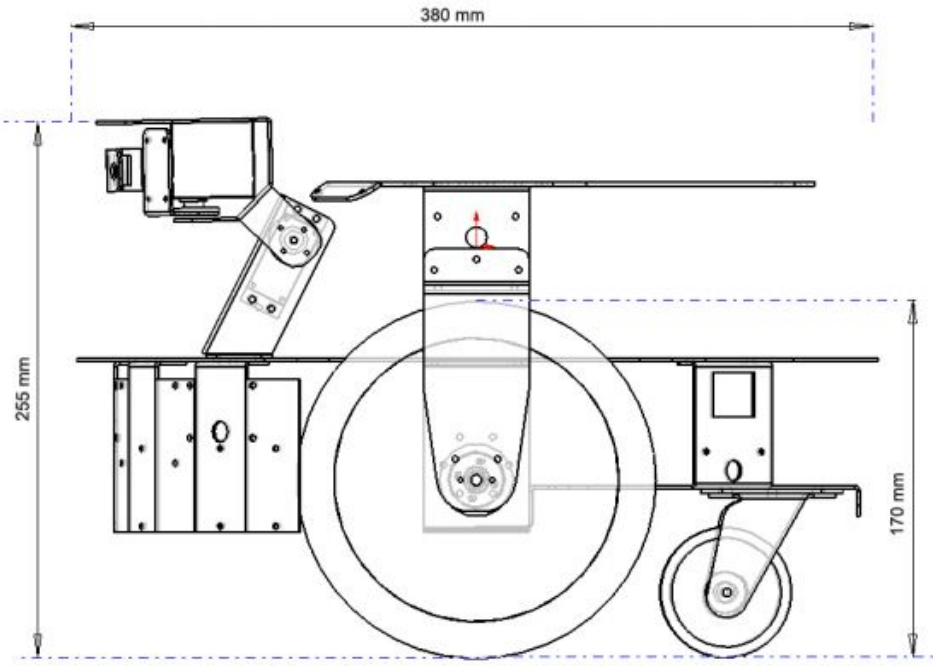

Fig. 2 X80 mechanical design

The SR300 camera mount CAD is shown in Fig. 3.

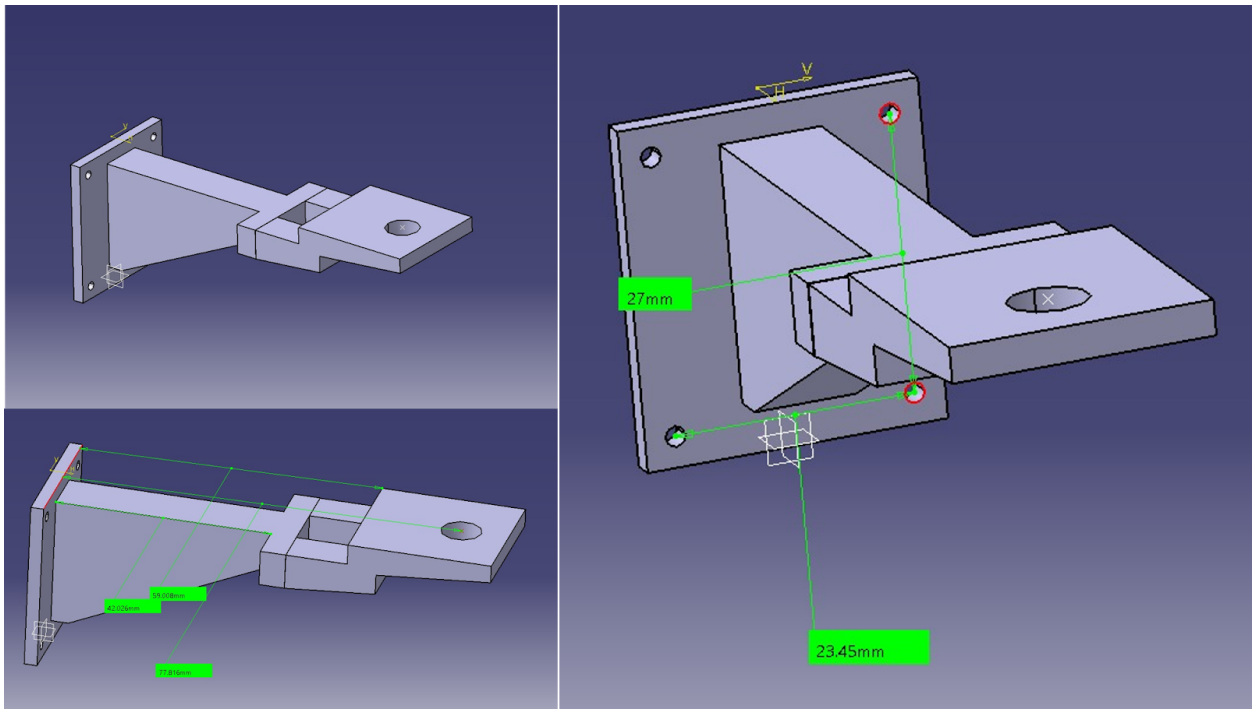

Fig. 3 SR300 camera mount CAD

Tests on mechanical implementation

Besides what has been already discussed, the DrRobot X80 robotic platform comes with a differential drive locomotion system that needs to work as expected for the robot to navigate correctly. To test the correct functioning of the motors and encoders the robot was teleoperated to move in both straight lines and to rotate around its own axis. An odometry node, which is better explained in the software section, was used to capture the robot movements, and RTAB-Map was used for visualization. The trajectories performed by the robot are shown in Fig. 4 plotted in blue.

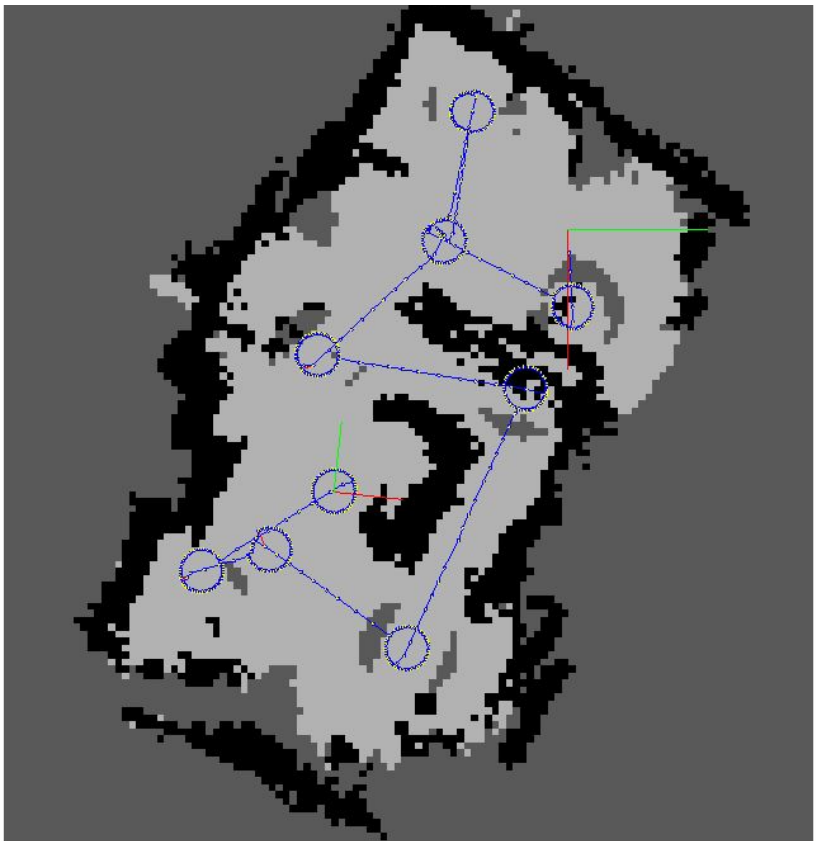

Fig. 4 Linear and angular trajectories performed by the X80 robot

The blue lines are the different positions of the robot captured at different times by the odometry node. It is clear that the robot by itself is able to trace straight trajectories, as well as to rotate around its axis with little error. This is essential for performing a precise autonomous navigation. The velocities of the robot were estimated as well, determining that the minimum linear velocity that the robot can move is 5cm/s and its maximum is almost 1m/s, while its possible angular velocities go from 0.1 to more than 2rad/s. It was later determined by the team that

the optimal navigation velocities are up to 10cm/s and 0.2 rad/s, which have been proved to be perfectly performed by the X80 locomotion system.

Electrical subsystem

Electrical design

The design aims to supply the complete system with a single battery with the intention of simplifying the operation. Having one power source reduces the risk of wrong connections after charging, inconsistent voltage levels for each subsystem and makes it easier to monitor the electric state of the complete system. With this in mind, a three-cell LiPo battery (11.1V) was chosen.

The main subsystems to supply are the robot embedded system and the on-board computer. The former were examined and an integrated regulator was found. This regulator has the part number LM2940S. Its datasheet reveals that it is a linear regulator with a maximum output of 1A and an input voltage ranging from 6V to 26V. Because this IC is already placed in its board and taking into account its maximum current, a separate regulator is used to supply the on-board computer (AAEON UP-Board [3]). The Pololu D15V70F5S3 buck converter admits inputs from 4.5V to 24V, has a selectable output and can deliver up to 7A of current. This output is perfect to supply the embedded computer that was in existence, the UP Board. According to its datasheet, its power requirements are a voltage of 5V and current of up to 4A.

The UP Board has USB and Ethernet (RJ45) ports, but no capability to communicate in a network through WiFi. Therefore, a USB WiFi adapter module was needed. The 3D camera can also be connected and fed through a USB 3.0 port, which the UP Board includes.

An electrical diagram of the system with its supply, connectors, ports and modules is shown in Fig. 5.

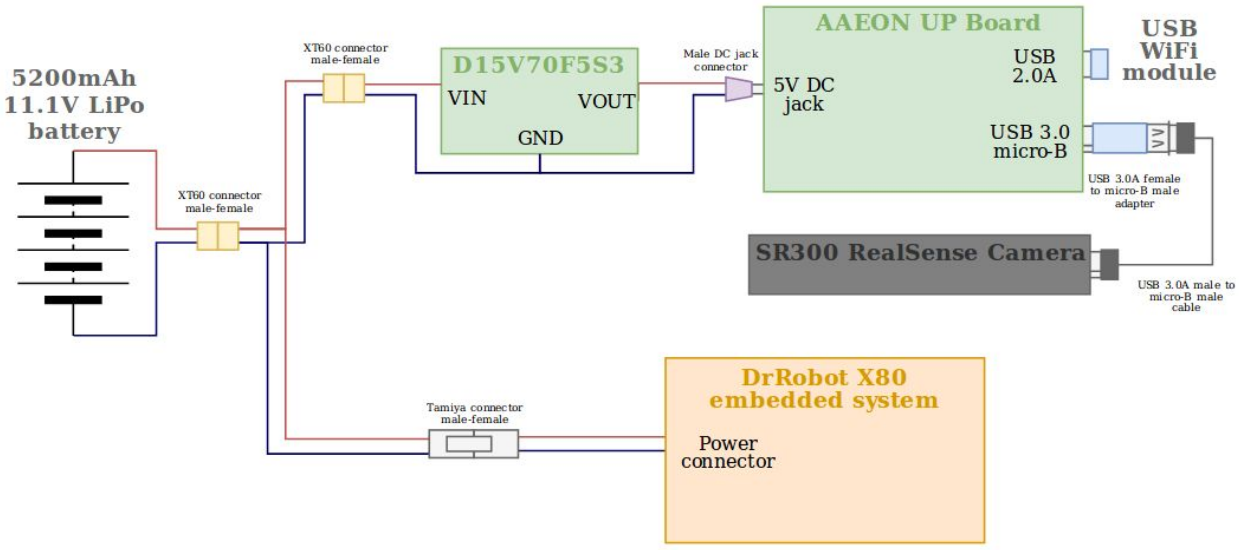

Fig. 5 Electrical design diagram

Electrical implementation

The electric supply subsystem was implemented with one Multistar 5.2 LiPo battery. The battery is secured to the lower deck of the robot and connected to an XT60 connector. The cable attached to this connector is shown in Fig. 6 and leads to a junction that was soldered to connect the robot controller and the onboard computer in parallel with their respective voltage regulators. Each of these components have the right connector to retrieve energy from the battery. The controller boards, for example, were originally fitted with a Tamiya connector, and this was kept and the other side of the cable was adapted, as seen below.

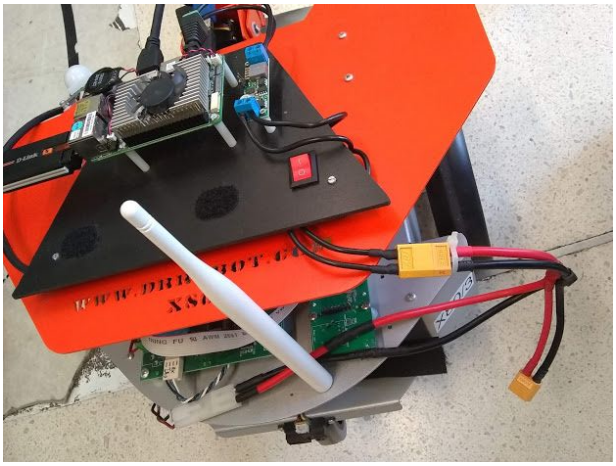

Fig. 6 Supply wiring. The battery goes on the bottom connector

The D15V70F5S3, seen in Fig. 7, is used to regulate the voltage for the UP Board to 5V. On the other side, the board included with the X80 is fed by the battery.

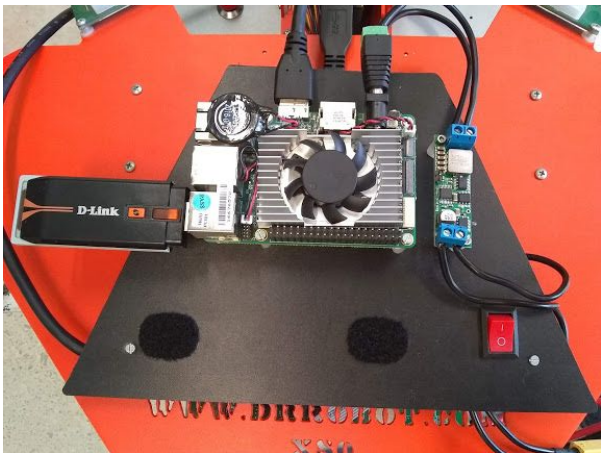

Fig. 7 UP Board supply with switch and regulator

The X80 embedded system's internal LM2940S regulator can take inputs up to 26V, which perfectly accommodates the battery. This regulator can be seen in Fig. 8.

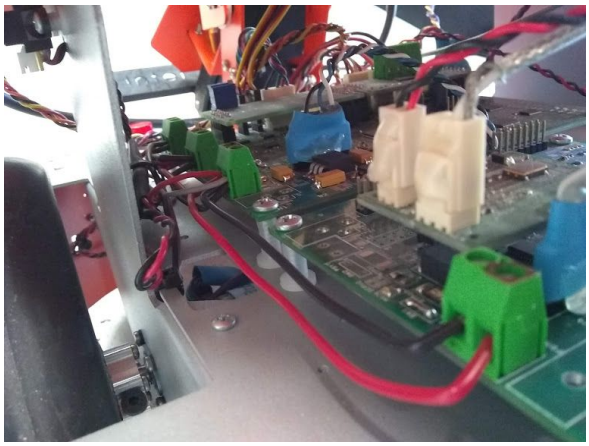

Fig. 8 Internal supply distribution. The regulator is the IC next to the blue connector

The USB WiFi module used was the D-Link DWA 140. This module was simple to install and allows the UP Board to see and connect to wireless networks as though it had an embedded wireless network interface card. With this configuration, the complete system is able to power on and function correctly.

Tests on electrical implementation

The electrical components of the system were tested in terms of voltage. The LiPo battery was ought to output a nominal voltage of 11.1V, which was measured using a voltmeter and found correct. It is important to mention that the voltage supplied by the battery oscillates between 12.6VDC and 10.7VDC nevertheless the functionality

of the system is not compromised because of the voltage regulators and also because the x80 robot regulator accepts this range of voltages and the on-board computer regulator also accepts this range of voltages. The D15V70F5S3 voltage regulator outputs a voltage of 5VDC as its data sheet states meeting the requirement in terms of voltage.

The battery voltage is also continuously monitored with a measuring device that includes three seven-segment displays and an audible alarm to alert for low voltages. It is shown in Fig. 9.

Fig. 9 LiPo tester

This gadget was connected all the time during testing. This made it possible to determine that, on average, one fully charged battery was enough to last for two hours before falling below a safety threshold of 10.5V. Thus, the available time of operation is defined as two hours. This is enough to perform many tests and move around a defined space multiple times, which makes the robot capable of working on its task without any worry to run out of power.

The operating voltage range of the battery was also found to be from 10.5V to 12.6V. This was determined during charge cycles. The charger takes batteries with a minimum voltage of around 10.1V and charges them up to 12.6V.

The current drawn from the battery is of around 0.8A from the robot controller circuit board and 1.4A from the UP Board regulator. This is well within the limitations of the battery, given its 10c discharge rating.

Although the robot includes an array of seven IR sensors, three ultrasonic sensors and two pyro sensors, it was found through software that some of them report a fixed distance measurement regardless of the presence or absence of an object, while the rest present a noisy behavior that would have to be filtered if a good use

of the sensors is desired. The final design omitted the mounted sensors and made use of encoder and camera information only.

Software subsystem

Software design

The system's software is based on topic publication and subscription to topics with code located in ROS nodes. A computer connected to the same network, but separate from the robot, initiates the process through a launch file. With this, each node in its corresponding computer gets initialized and the system starts working. Information about the 3D environment is taken from the camera to collect a point cloud of the environment. The UP Board, through a different ROS topic, sends motion commands to the X80 in a format that is understandable by the embedded controller. This is done through a library published by Dr Robot. The UP Board also needs to keep a register of the state and actions of the robot to aid its localization through dead reckoning. Additional information from the map can later be used as well. While all this happens, the result of the map can be visualized in the remote computer with the help of the Rviz software. A behavioral flow diagram is presented in Fig. 10.

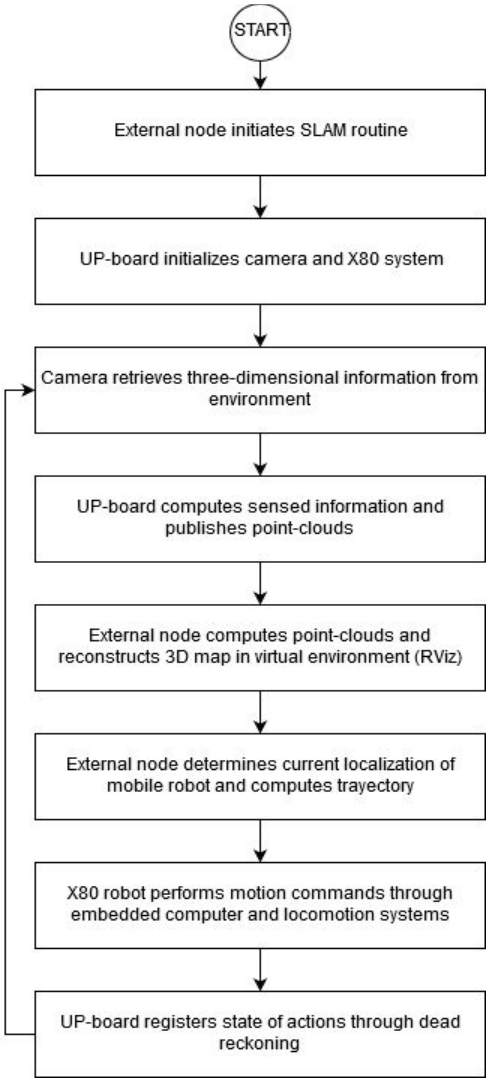

Fig. 10 Behavioural flow diagram for the navigation system

The first general proposal for inter-device communication and ROS topic ideas is included in Fig. 11. This diagram depicts the wireless connections between the nodes on the system, which would implement a fully integrated WiFi system with dual serial communication channels supporting both UDP and TCP/IP protocol.

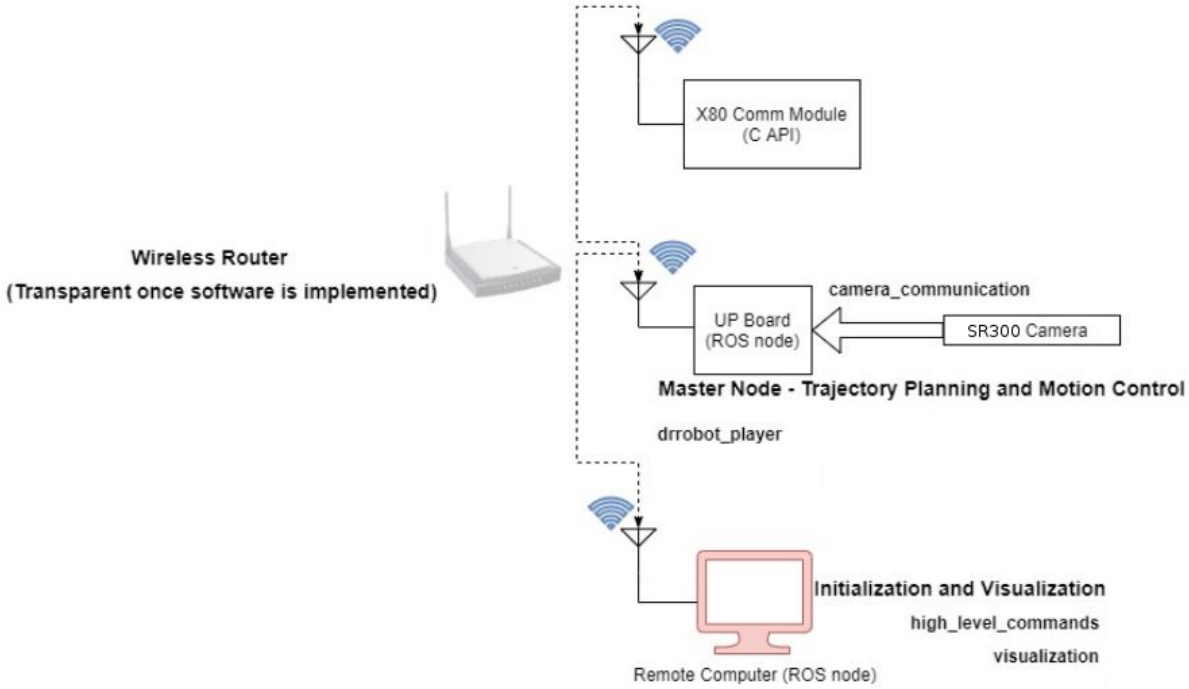

Fig. 11 Original design of ROS nodes and topics for message passing through WiFi

Software implementation

To ensure connectivity through a wireless network, a Trendnet N300 wireless home router was used. This helped isolate traffic used for the project from all the network traffic in the campus and define static IP addresses so that ROS nodes could have a fixed master URI. Furthermore, the X80 looks, by default, for a network with an SSID of “dri”. The router offered access to configuration options through a browser so that static DHCP could be set. On the other hand, the IP address of the master node was set in the *.bashrc* files of each device so that communication was always straightforward.

A more thought out, but still simplified, diagram of nodes and topics is shown in Fig. 12. This is a more recent construct, created with clearer ideas about the data interchange necessary in the system.

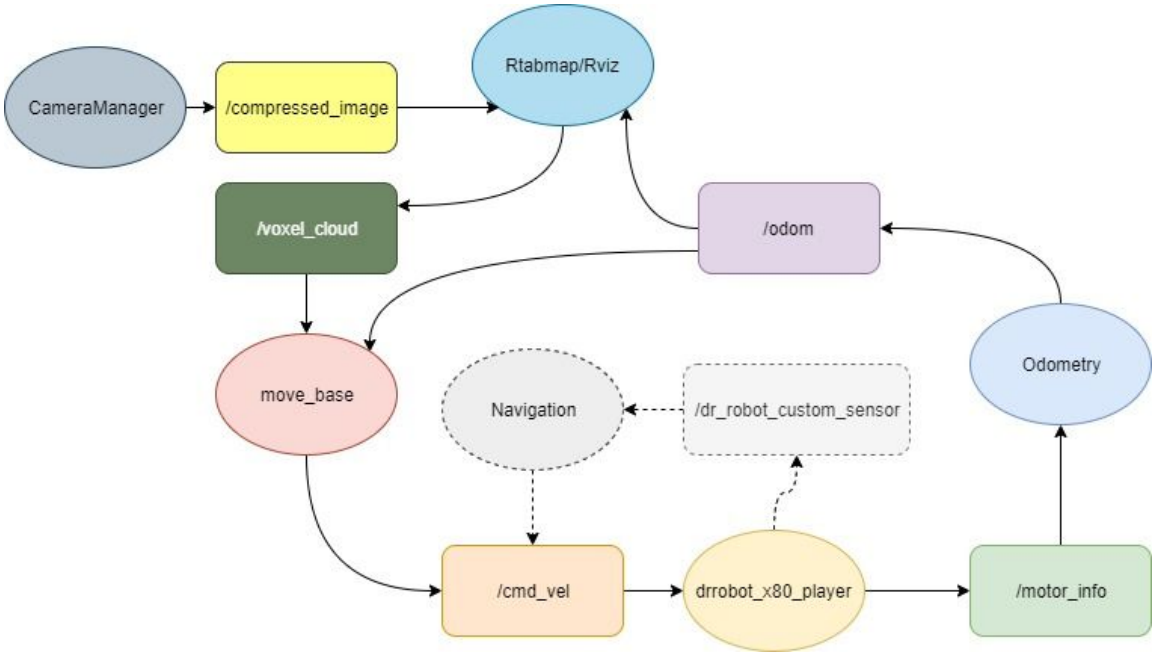

Fig. 12 Simplified node and topic implementation diagram

The drrobot_x80_player node is responsible for wireless communication with the robot. It sends the desired velocity commands (and can adjust other things such as servo angles) encoded in the way the manufacturer established for the X80. It also receives data from the robot's sensors and publishes it into various topics.

An important parentheses must be made regarding the functionality of the drrobot_x80_player node code. The library to make the robot compatible with ROS was created by the manufacturer, who published it on Github in 2015 and has not been updated since its first commit, as of December 2019. The required libraries are *drrobot_X80_player* and *DrRobotMotionSensorDriver*, which can be found at https://github.com/gitdrrobot. However, when the X80 is operated through this library, all velocity commands are ignored. This was found to be caused by the *DrRobotMotionSensorDriver* code not considering the controller board called "X80SV" when it is asked to assign a message destination ID to the robot. The error is found on line 763 of the original DrRobotMotionSensorDriver.cpp:

```
if ((_robotConfig->boardType == I90_Motion) || (_robotConfig->boardType ==
I90_Power) || (_robotConfig->boardType == Jaguar))
  {
    _pcID = COM_TYPE_PC;
    _desID = COM_TYPE_MOT;
  }
  else
  {
```

```
    _pcID = COM_TYPE_PC_PLUS;
    _desID = COM_TYPE_MOT_PLUS;
  }
```

This line does not consider the possibility of having a board type of X80SV, which is the one passed by the player node. Our current fix comments out that fragment and just makes sure that the variable _desID is assigned the constant COM_TYPE_MOT (with a value of 0x01) instead of COM_TYPE_MOT_PLUS (with a value of 0x8B). With this change, the robot identifies the message correctly and is able to move.

Originally, the IR sensors would be read from the drrobot_custom_sensor topic and their data would be used in a Navigation node to calculate an appropriate linear and angular velocity for the robot. This approach has been left behind. The new navigation strategy uses the data from the generated map and should, by design, work as follows:

An Odometry node subscribes to the motor_info topic, where the robot's encoder values can be found. It then calculates the displacement and velocity to know how the robot is moving and where it is located with respect to its starting position at any time. If the displacement of each wheel and their current velocity is determined (by measuring time and encoder count), then the displacement of the robot in the last sample interval can be found using:

$$d = \frac{d_R + d_L}{2}$$

$$o = \frac{d_R - d_L}{b}$$

Where d is the linear distance, o is the angular displacement, b is the distance between the two wheels and d_R and d_L are the linear displacements that the right and left wheels moved respectively. With this information, the robot's position in the X-Y plane can be updated by adding the displacements to stored x and y variables. A θ variable is also stored to know the orientation of the robot.

$$x_i = x_{i-1} + d\ cos(\theta + \frac{o}{2})$$

$$y_i = y_{i-1} + d\ sin(\theta + \frac{o}{2})$$

$$\theta_i = \theta_{i-1} + o$$

This data can be encapsulated in a nav_msgs/Odometry message and published to an odom topic. This information, along with the image and depth data obtained from the camera, are taken by RTAB-Map [6] to construct a three dimensional map of the robot's environment. The odometry data is used to know the location and orientation of the robot and the depth and RGB data is used to detect objects, perform feature recognition and present the map in a screen to the user. Note that the images are compressed before transmission to make the processing and communication quicker.

RTAB-Map is then required to publish its point cloud so that another node, move_base, can use it along with the odometry. This node is part of the ROS packages and is used for navigation. By getting the map and current position data, it should be able to find a route with the lowest cost to certain destination and send the necessary velocity commands back to the drrobot_x80_player node. Thus, the loop is closed.

As stated in the software design the robot's software is based on topic publication and subscription to topics with code located in ROS nodes. To visualize and understand the nodes structure of the system tf can be used. Tf is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time. tf can operate in a distributed system. This means all the information about the coordinate frames of a robot is available to all ROS components on any computer in the system. There is no central server of transform information. There are essentially two tasks that any user would use tf for: listening for transforms and broadcasting transforms.

Anyone using tf will need to listen for transforms:

- Listening for transforms - Receive and buffer all coordinate frames that are broadcasted in the system, and query for specific transforms between frames.

To extend the capabilities of a robot it is needed to start broadcasting transforms.

- Broadcasting transforms - Send out the relative pose of coordinate frames to the rest of the system. A system can have many broadcasters that each provide information about a different part of the robot. [7]

In Fig. 13 the ROS tf tree about the system is presented. rqt_tf_tree provides a GUI plugin for visualizing the ROS TF frame tree. In the system the tree must be unified

for its correct functionality and to avoid errors and delays in the frame transmission.

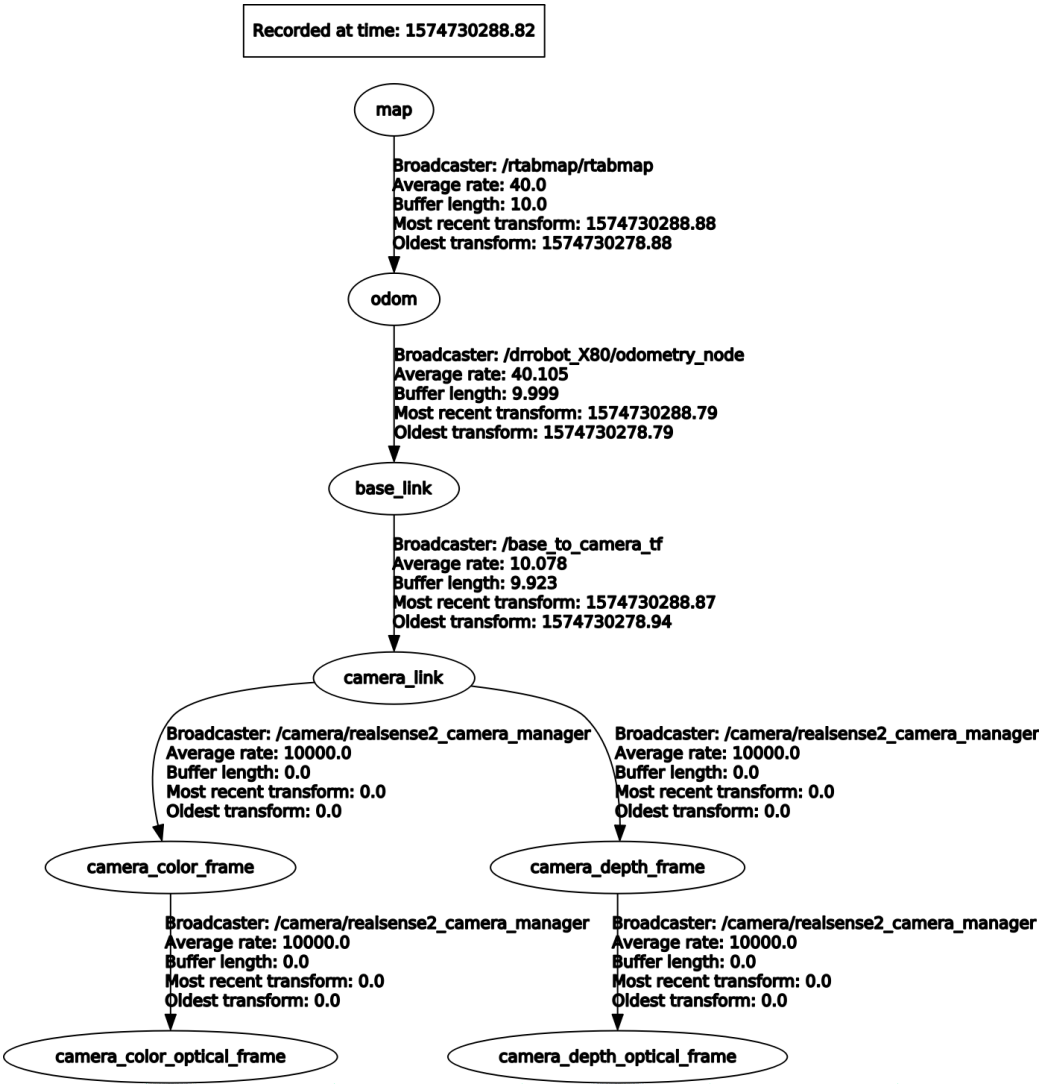

Fig. 13 rqt_tf_tree depicting the transforms between frames

In Fig. 14 the rqt_graph can be visualized. rqt_graph provides a GUI plugin for visualizing the ROS computation graph. The 4 most important components camera, rtabmap, drrobot_X80 and move base are connected by specific nodes showing the relation between them. Also the odometry node is presented and it can be seen that it publishes its data in the rtabmap component. The objective of the move base component is to create the global cost map and local cost map for the autonomous navigation.

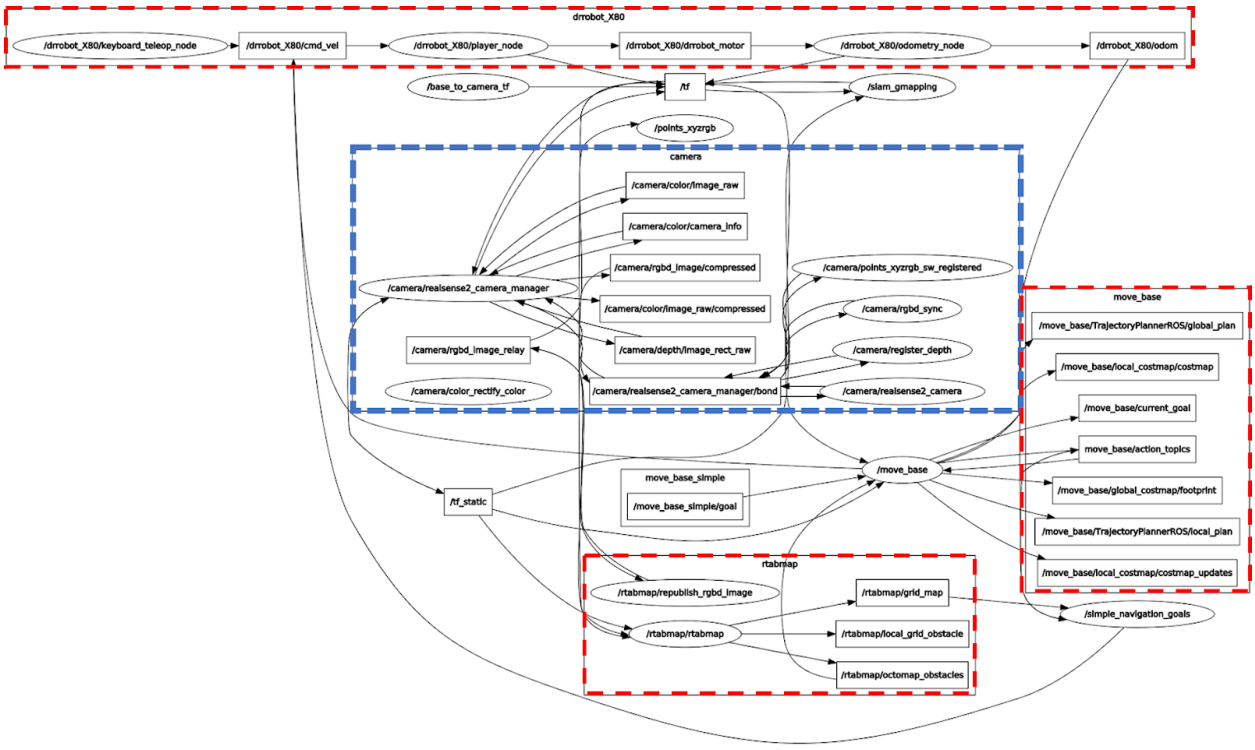

Fig. 14 rqt_graph depicting the groups (large rectangles), active nodes (ovals) and topics (small rectangles). The blue rectangle runs on the UP-Board and the red rectangles run on the external PC

What follows is a list of important files that have been generated and a brief description of each one. The list includes launch files that run nodes and programs, yaml files with configuration parameters and C++ files with specific code and functions.

DrRobot X80 nodes:

- drrobot_player.cpp: This file was created by the X80's manufacturer [10] and is used to make it compatible with ROS. It subscribes to a cmd_vel topic to move the robot and publishes to various topics with sensor information. It runs in the UP Board.
- drrobot_keyboard_teleop.cpp: This file takes keyboard inputs from a computer and sends corresponding velocity messages to the cmd_vel topic so that drrobot_player can read them. It has been used as the main motion control before achieving autonomous navigation. It runs in the controlling computer.
- drrobot_odometry.cpp: This file defines a node that subscribes to the motor_info topic so it is able to get encoder data. It also computes the motor rotation and wheel advance distance for each cycle and calculates the

displacement of the robot from its origin, as well as the linear and angular velocities needed to create valid odometry messages.

- send_goals.cpp: This node is responsible for sending autonomous goals to the move_base node (i.e. the navigation node) based on the current 2D gridmap generated by RTAB-Map. Also, it performs the finite state machine behaviour explained further in this document.

Launch files:

- sr300_throttle.launch: In this file a line was introduced using the tf package to specify a static_transform_publisher to link two nodes. This was done in order to unify the tf tree of the system. The base_link was staticly linked to the camera_link in a rate of 1 Hz, with a translation of [0.16 0 0.25]m in the XYZ axes correspondingly, and a pitch rotation of 0.28rad for the camera is facing slightly downwards to capture pointclouds from plane surfaces. This launch file also initializes the SR300 camera, while synchronizing and publishing RGB-D (color and depth) data through rgbd_sync to reduce bandwidth consumption and latency.
- x80_slam.launch: This file launches three main parts of the ROS system, the three X80 nodes explained above, RTAB-Map with the visualization parameters configured for RViz, and the navigation nodes. All of these nodes are run in the external PC to subscribe to camera topics using a mechanism named rgbd_relay, to the frame transform published by the UP-Board, and to send navigation messages to the X80 platform.
- send_goals.cpp: Although it is not a launch file, this node is mentioned here as it needs to be run independently from the other nodes to receive important in-console information about the current state of the autonomous navigation behaviour.

Configuration files:

- Four costmap_params.yaml files: These files are used to define parameters for the navigation node. They mainly define minimum and maximum speeds for the autonomous controller, obstacle height limits, map resolutions, distances to keep away from obstacles, sensor sources, frames of reference and the type of map to create. They are used to create cost maps, which contain data needed to find a low cost path to a goal. A global map is defined to keep track of the layout of the room and a local map is used to better define nearby obstacles. They were mainly based on a tutorial by the Federal University of Technology - Parana [11].

Tests on software implementation

The test on the software implementation were functional and performed as an iterative process.

Odometry: The odometry program was developed using Visual Studio Code and tested with the robot in the field using the Terminal as a monitor of the results. The expected results were to find the correct displacement in the correct direction when teleoperating the robot. The test consisted in tracing a closed path with the robot, going to some corners of the room, performing some turns and finally returning to the same initial position. A spot on the floor was marked to identify the origin. The dead-reckoning odometry accumulated an error that is visible in the image: the end position has an absolute error of 14.1 cm in the x axis and -39.2 cm in the y axis.

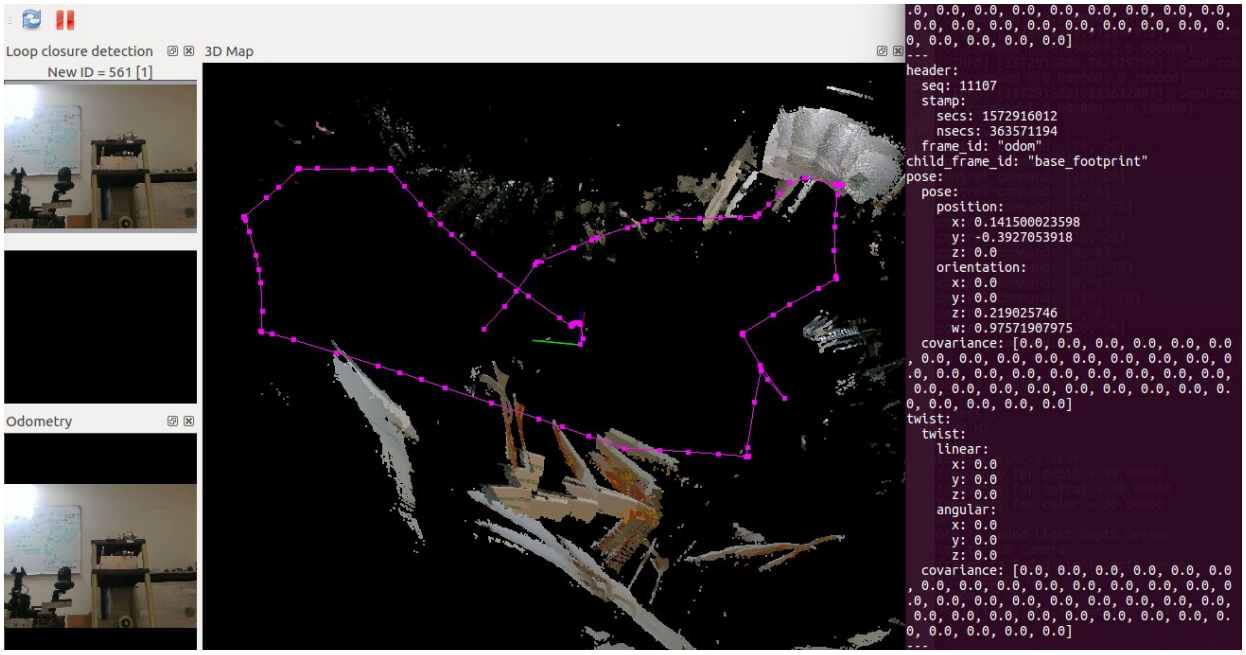

Fig. 15 Odometry error test

During this test, however, only the odometry coming from the encoders was used. RTAB-Map includes a feature where the visual information can be used as a second input to determine the current position of the robot. When a location is known and a point cloud exists, features of the image that have been seen before can be identified. This then leads to the odometry error being reduced. This is known as loop-closure in RTAB-Map.

Obstacle detection: The main objective of the project is to obtain a visualization of the environment in three dimensions. However, to achieve this, it is important to

detect areas of the space where objects higher than some threshold are located and categorize them as obstacles. The information gathered by the robot can be used not only to build a 3d representation, but also to create a 2d grid with cells that are either clear path or obstacles. With a testing layout shown in Fig. 16, the robot was driven through the small maze and the resulting map and grid obtained.

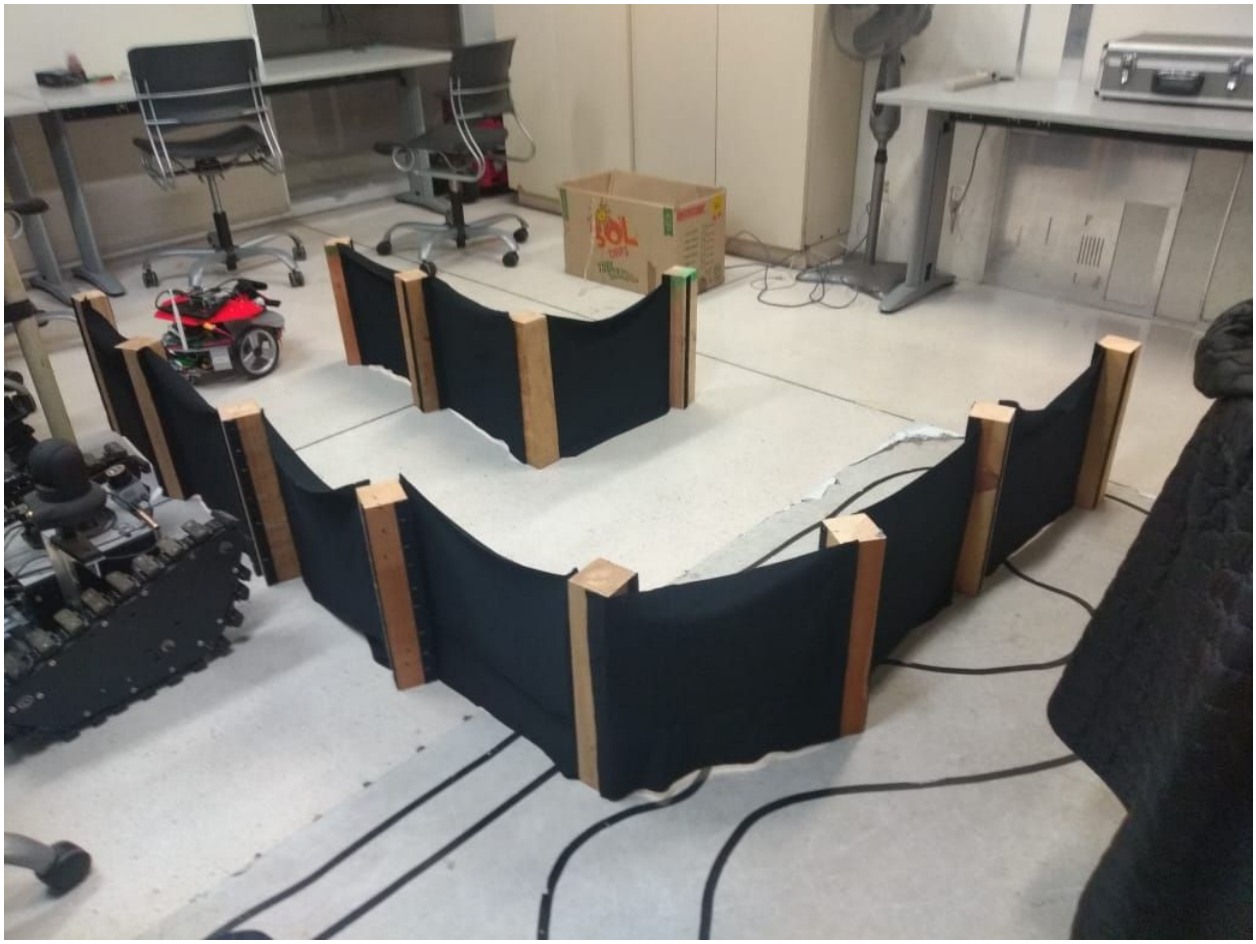

Fig. 16 Layout for testing obstacle detection

As the result is not numerical and there is not a defined method of evaluation of the cloud map the test at this point of the development of the project stays subjective. The system was able to construct a map being teleoperated and the result compared to the layout can be considered acceptable. It is possible to make this statement observing the original image and the constructed image. The odometry works fine in terms of providing the information about the position of the robot, the images are used to construct a coherent representation of the environment because all the objects that the robot encountered were detected. Some points in the cloud map that are not part of the layout are introduced by the noise of the

pictures taken by the camera given the different light brightness in the room or misclassification of the depth of the objects around the robot.

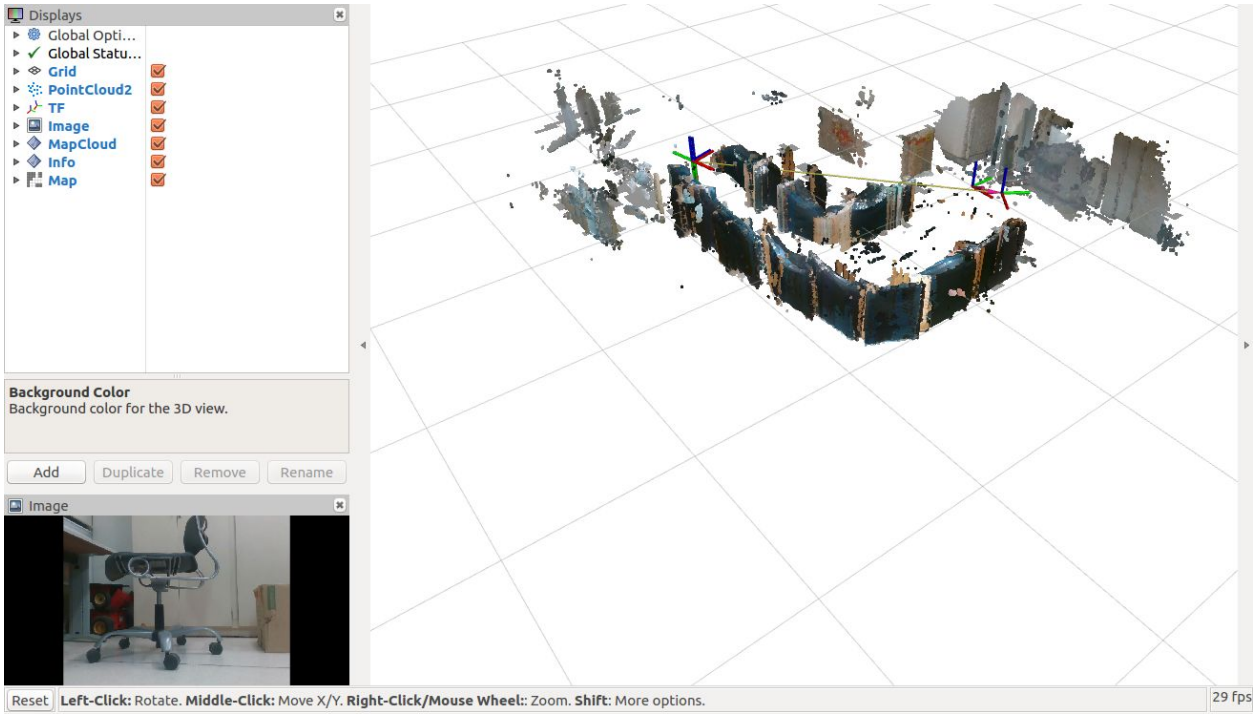

Fig. 17 Resulting cloud map in three dimensions

In this test, the distinction between clear areas and obstacles is made and the general shape of the maze structure can be seen in the grid map in Fig. 18. Possibly due to the lack of valid points and loop closure in all the frames received, some features seem to be duplicated and displaced by some centimeters. The accuracy might be possible to increase by varying the speeds and using more recognizable features for the images taken, but the test was still considered successful.

Fig. 18 Resulting grid map

A second test of similar nature was performed later on, this time taking more advantage of the loop-closure detection and odometry correction from RTAB-Map. To achieve this, once the robot had navigated to a new area it performed a full rotation around its own axis to be able to complete a loop-closure, thus rectifying its position and that of the most recent pointclouds. The 3D cloudmap achieved from this test is shown in Fig. 19.

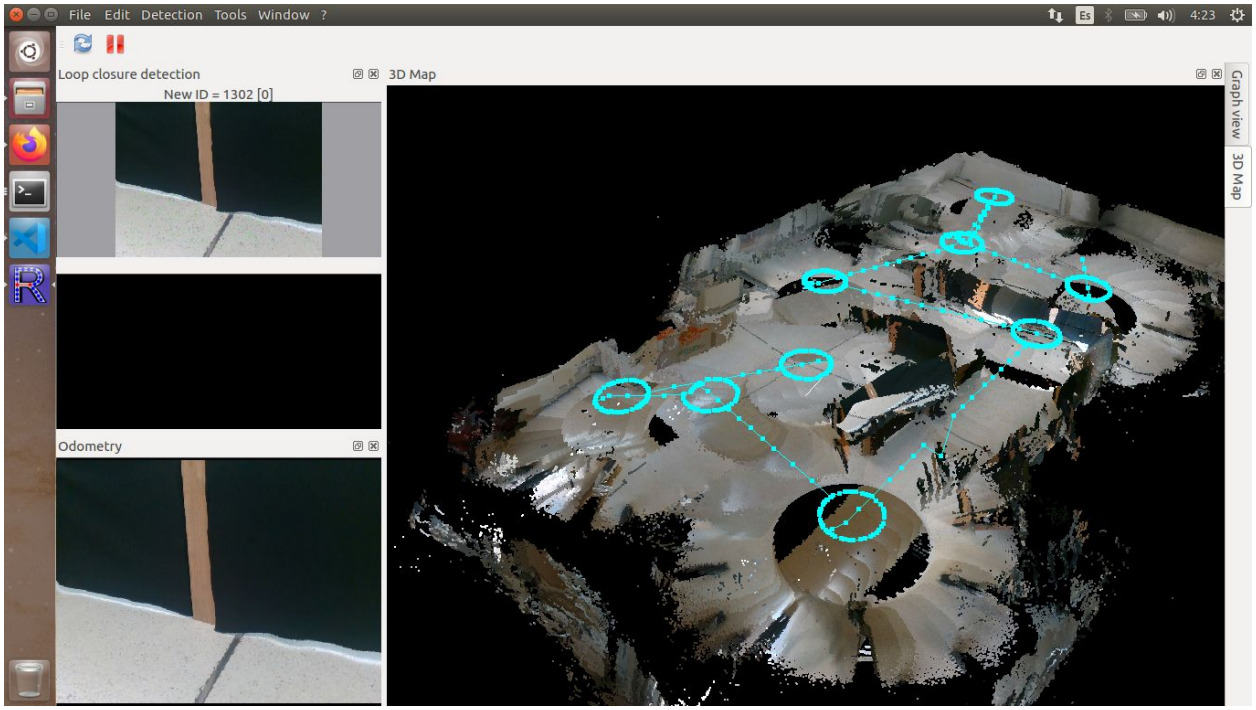

Fig. 19 3D cloudmap from the mapping test

Once again, RTAB-Map is able to tell obstacles from plain surfaces, depicted in black in the generated 2D gridmap. This is shown in Fig. 20, which resulted in being precise than previous iterations of this test.

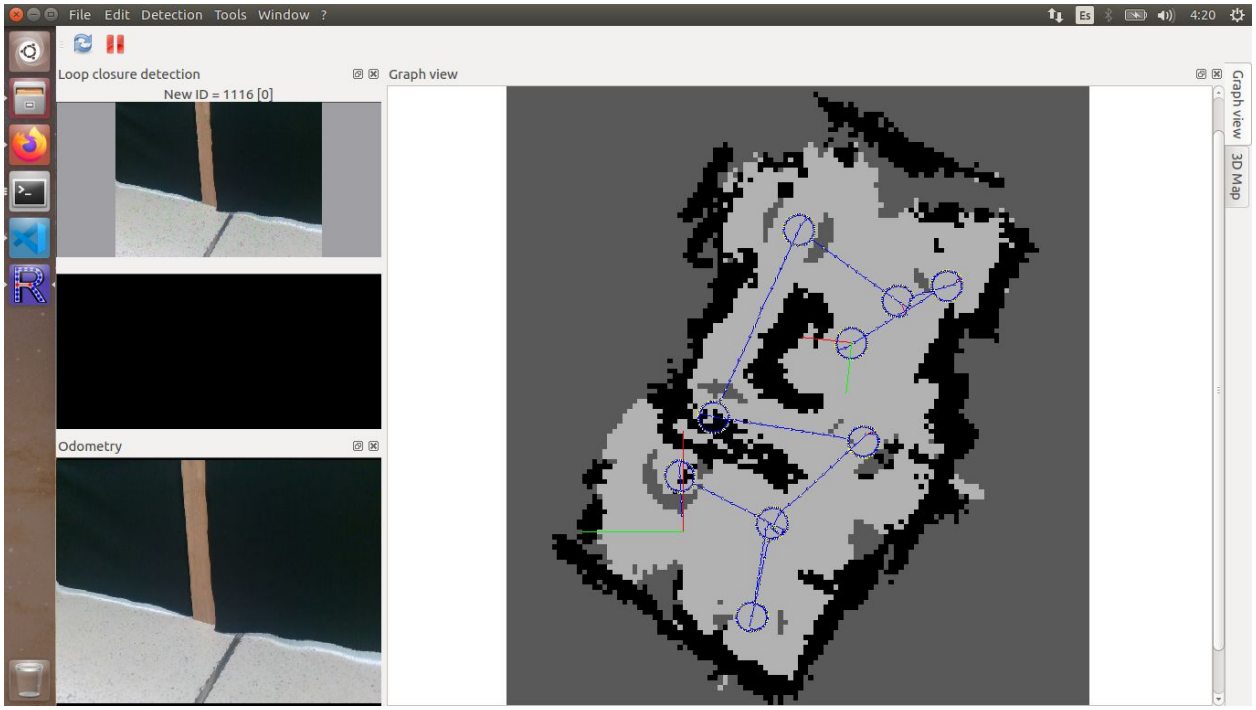

Fig. 20 2D gridmap from the mapping test

Costmap construction: The costmap is a feature used by the navigation stack. It locates obstacles and transitable paths and assigns a cost to each possible route from the position of the robot to a defined goal. This way, it is possible to determine the best path to follow. Points seen in the pointcloud are projected down to be identified as obstacles.

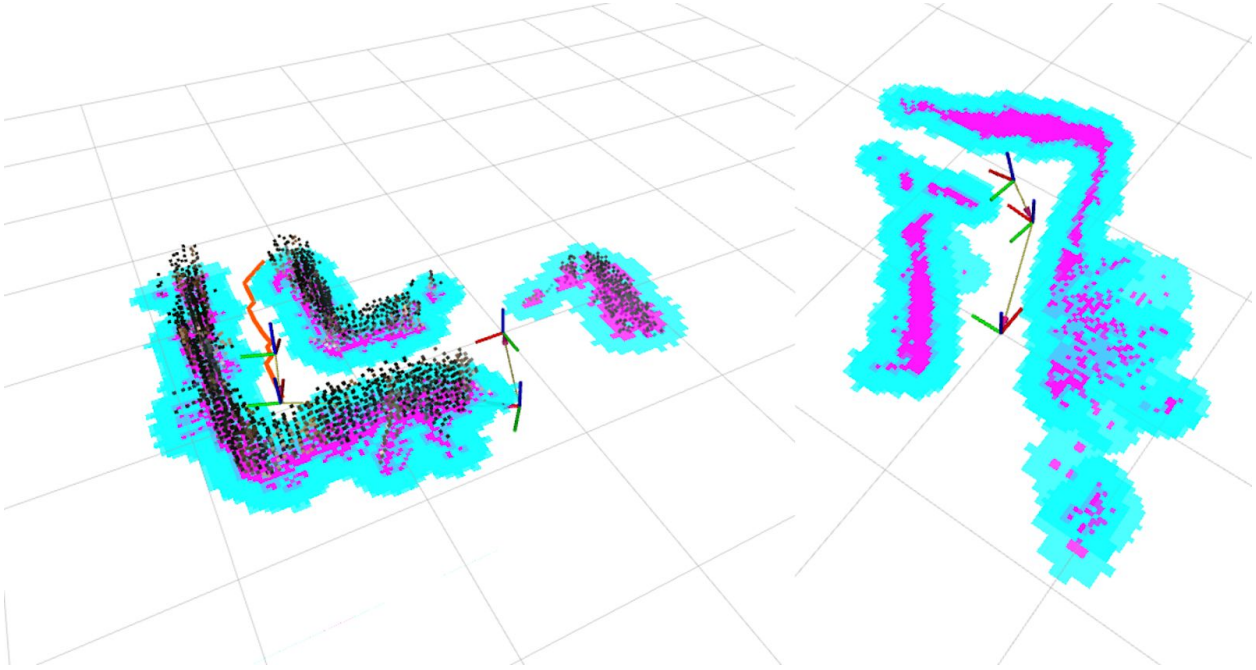

Fig. 21 Creation of 2D costmap with inflation

Through yaml configuration files, it is possible to define a radius for the robot from its center and a clearing distance from the obstacles (seen in blue). After a variety of tests, it was determined that the best results were obtained by reducing the radius of the robot and increasing the inflation radius for the obstacles. This way, the robot would not consider itself blocked by narrow paths seen incorrectly by noise and would avoid obstacles by choosing a route far from them.

Navigation with manual goals: With the information from the costmaps, the navigation stack [8] was able to get goals and navigate the robot to them. The tests were done in RTAB-Map with empty maps. After setting a goal with the GUI, the planned trajectory appeared on the screen and the robot navigated to the goal. Some strange behaviors were found. First, the robot would usually get stuck and consider that there was insufficient space to pass through gaps that were actually wide enough. This was solved by tuning costmap parameters. The other behavior was the way that the robot followed its projected trajectory. It usually did not follow the line in a straight path, but rather traced a curve that started and ended

on the planned line. This was deemed useful because the rotation sometimes helped the camera see a bigger part of the environment. Although the reasons for this behavior are still unknown, it might not be easy to modify, since the navigation stack manages its own feedback algorithms.

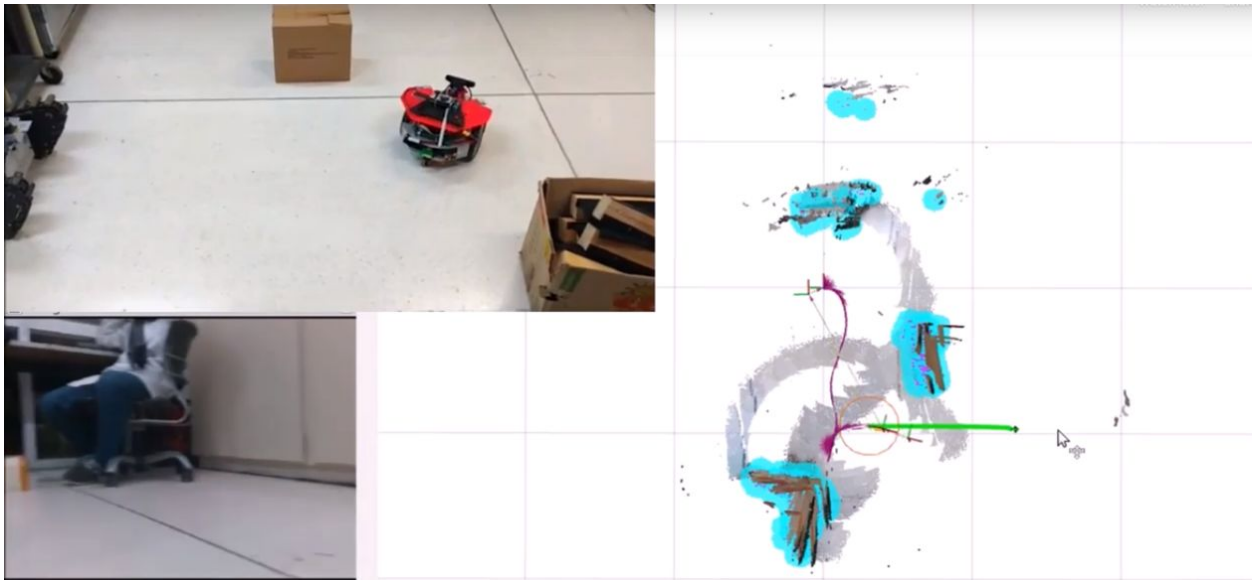

Fig. 22 Robot following planned trajectory after a manual goal is set

A tolerance was added later so that the robot would consider to have reached the goal if it was inside a radius centered around its objective and in an almost free orientation. This worked well for the final implementation, since it made it more agile.

Autonomous goal selection: Various approaches were taken to look for a good algorithm that would ensure the complete room was explored. The first approach was to send a goal to a location far from the origin, ideally out of the bounds of the room. The robot would then try to get there in a straight line and map any obstacles it met. It would finally get to the boundary of the room and determine that a route was not possible through there. A new path would be planned that would pass through unexplored areas. Eventually, the robot would find that this path was not a way either and would calculate another route. This would repeat until the robot had seen at least all the limits of the room and the process aborted due to there not being any possible way to arrive at the destination. A test using this technique is shown in Fig. 23.

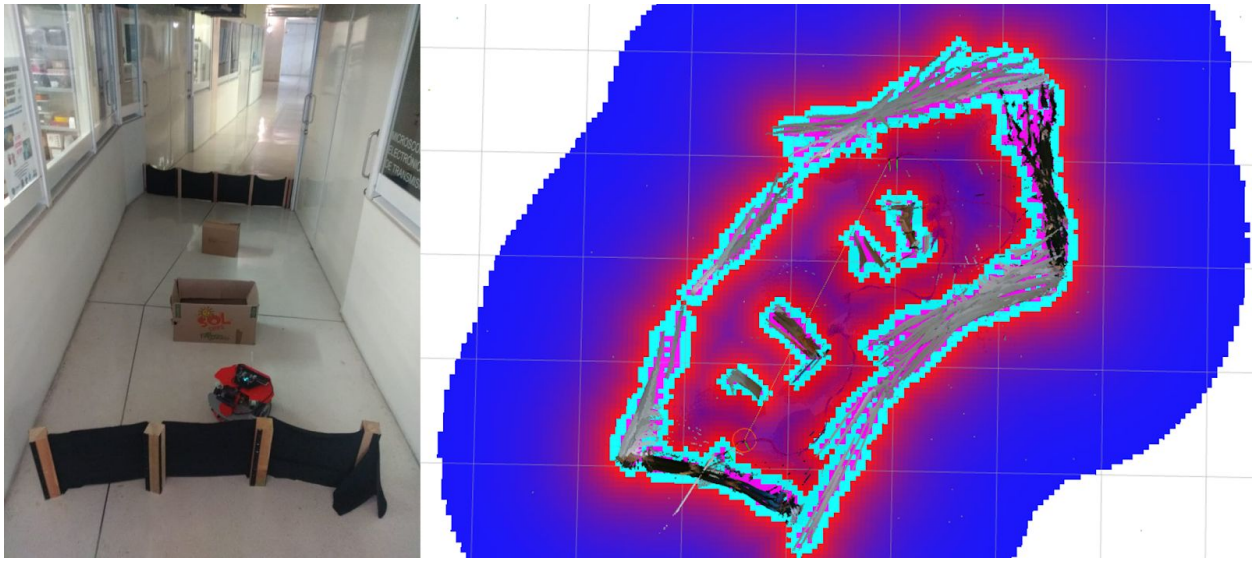

Fig. 23 Test for the “goal at infinity” approach. The result is warped, but all the limits are mapped

The limits of the space are all shown in the map, but it is distorted and turns a straight environment into a curved one. The probable reason for this is the lack of loop closures from the camera images. The robot did not return to its origin since it is not part of the algorithm. Also, as seen in Fig. 24, a white, unexplored spot can be observed. The robot was able to see the far wall without travelling through the unexplored region and had enough information to know that a path to the goal would not be possible through that wall. Thus, the white region did not even need to be explored.

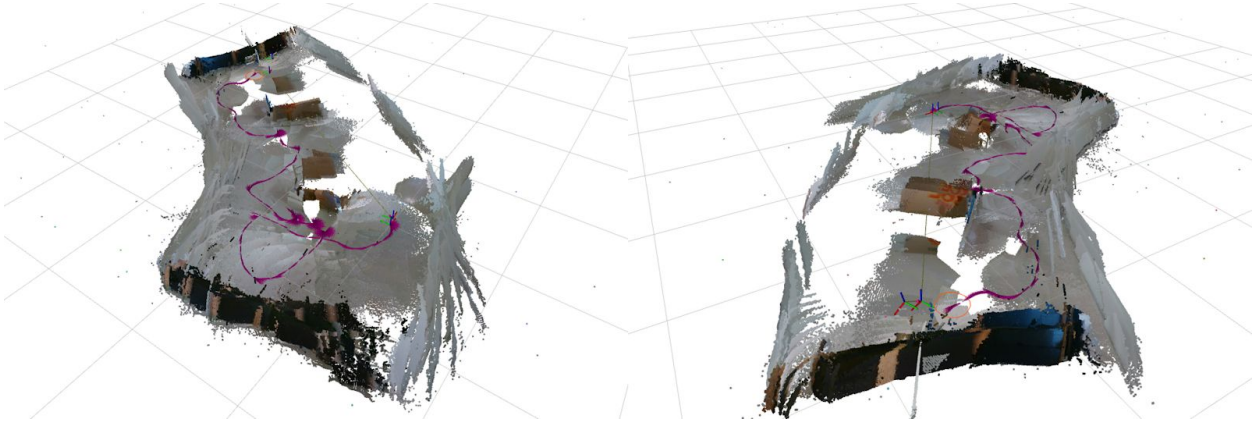

Fig. 24 Resulting pointcloud of the “goal at infinity” approach

The next attempt made use of a matrix representation of the grid map, where each point corresponds to an element of a square matrix. A value of -1 means unexplored, 0 corresponds to flat ground and 100 is an obstacle. The first use of

this matrix consisted in adding the values from neighbors multiple times to reduce unknown clusters and then choose the one with the resulting biggest value. The result was unsatisfactory because the initial map is always unknown, which means there is only one big cluster, which is also the biggest. The goal is set at the corner of this cluster due to the logic of the program. Therefore, the goal with this approach always goes far away. It is also easy to get stuck trying to go to an unreachable goal.

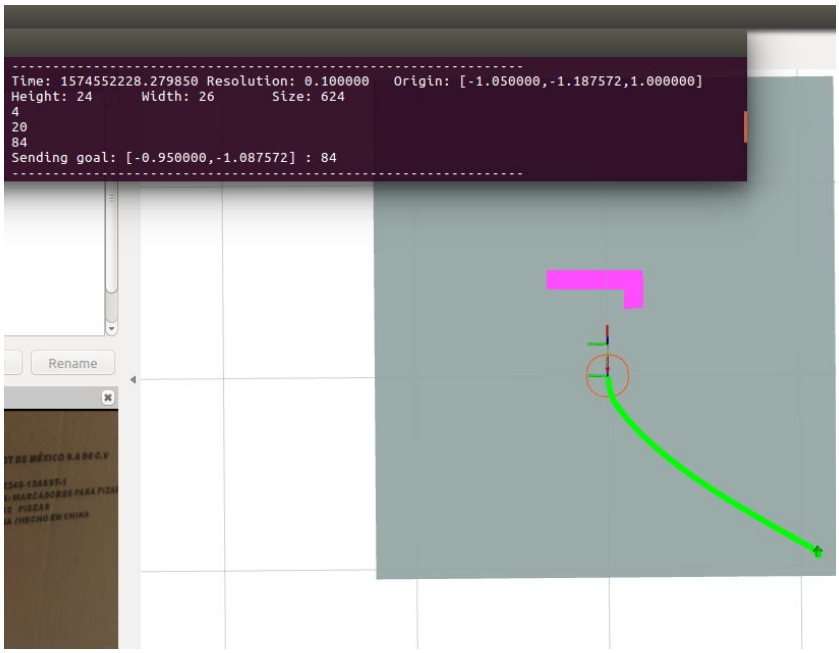

Fig. 25 First attempt at choosing the point from matrix processing. The corner is always chosen

The last approach chooses a point closer to the robot so that the map gets expanded gradually, but also defines a minimum radius from the robot so that it at least makes some progress. This is important because the points directly below the robot are unseen by the camera and will remain unexplored unless the robot moves and looks at them. This approach also uses a state machine that makes the robot choose a goal, go to it, perform a full rotation to scan that point's surroundings and obtain loop closures and then returns to the origin to correct for odometry errors. The state machine is seen in Fig. 26.

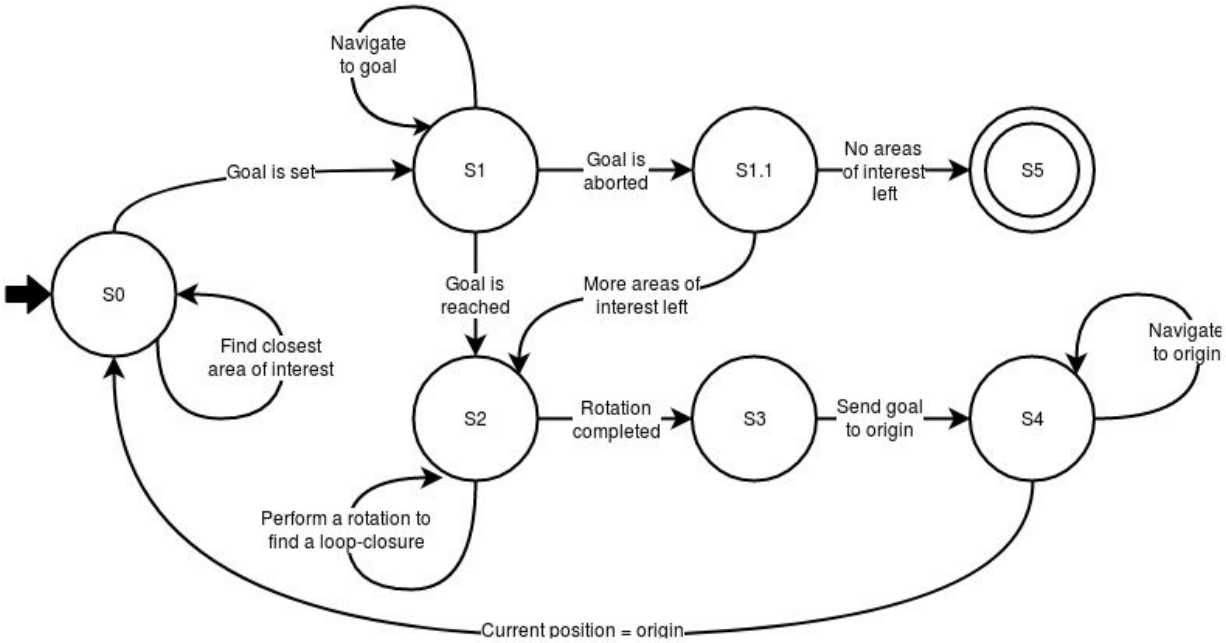

Fig. 26 Implemented finite state machine

A map made with this approach of a small enclosed space is shown in Fig. 27.

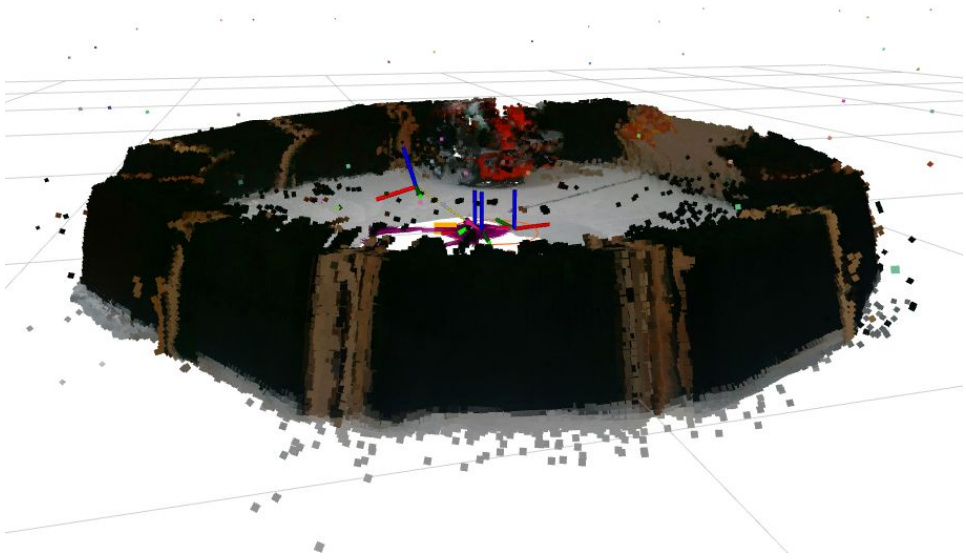

Fig. 27 Map made with minimum radius approach

A condition was set so that, once the ratio of unexplored to total points dropped below a threshold, the program stopped and the room was considered mapped. This, however could not be tested due to time limitations.

Conclusions and future work

After having implemented and tested the mechanical, electrical and software subsystems documented throughout this document, the team is confident in the correct functioning of the robotic system. Even though the system is already at a high level of integration and functioning, it is recommended to keep developing algorithms to ensure a more complete and accurate mapping of the room, as well as the tuning of already existing parameters to reduce noise, decrease the error from the odometry, ensure more frequent loop closures and make the complete process more efficient. We consider that the main area of opportunity of the current implementation is to design a more robust algorithm for setting navigation goals autonomously, since the current one is enough but can be greatly improved with the use of more sophisticated ways to determine the most relevant cluster of unknown grids within the 2D gridmap.

Although the project was developed in ROS Kinetic Kame running on Ubuntu 16.04, it is recommended that any new contributor makes the effort of migrating the current state of the project to the newest ROS distribution Melodic Morenia and Ubuntu 18.04, as it will guarantee a longer support and the potential implementation of newer packages that might help push the project forward in the future.

The GitHub repository containing all the source code and documentation generated for this project can be found in this link:

https://github.com/A01371852/ROS_Autonomous_SLAM

The README.md file contains a tutorial for replicating the current state of the project. Also, there are a number of closed issues that discuss common problems that were encountered throughout its development. Also, the directory *Bibliography* contains a collection of relevant documents and papers that can be very helpful for future contributors to this project.

References

[1] Intel Corporation. (2016). *Intel RealSense Camera SR300*. Retrieved from: https://www.mouser.com/pdfdocs/intel_realsense_camera_sr300.pdf

[2] Dr.Robot. (2006). *WiRobot X80 USER MANUAL*. Retrieved from: https://www.cs.princeton.edu/courses/archive/fall11/cos495/X80_Manual.pdf

[3] AAEON. (2019). *UP Datasheet V8.5*. Retrieved from: http://data-us.aaeon.com/DOWNLOAD/2014%20datasheet/Boards/UPDatasheetV8.5.pdf

[4] PololuCorporation. (2019). *Pololu Step-Down Voltage Regulator D15V70F5S3*. Retrieved from: https://www.pololu.com/product/2111/specs

[5] Texas Instruments. (2014). *LM2940x 1-A Low Dropout Regulator*. Retrieved from: http://www.ti.com/lit/ds/symlink/lm2940-n.pdf

[6] Open Source Robotics Foundation. (2019). *rtabmap_ros*. Retrieved from: http://wiki.ros.org/rtabmap_ros/

[7] *Idem*. (2017). *tf_ros*. Retrieved from: http://wiki.ros.org/tf

[8] *Idem*. (2019). *navigation*. Retrieved from: http://wiki.ros.org/navigation/

[9] Maxwell, R. (2013). *Robotic Mapping: Simultaneous Localization and Mapping (SLAM)*. Retrieved from: https://www.gislounge.com/robotic-mapping-simultaneous-localization-and-mapping-slam/

[10] Dr.Robot. (2015). *gitdrrobot/drrobot_X80_player*. Retrieved from: https://github.com/gitdrrobot/drrobot_X80_player

[11] Fabro, João & Guimarães, Rodrigo & Oliveira, André & Becker, Thiago & Brenner, Vinícius. (2016). *ROS Navigation: Concepts and Tutorial*. Retrieved from: https://www.researchgate.net/publication/302986850_ROS_Navigation_Concepts_and_Tutorial