Development of new 2D and 3D navigation and exploration strategies for a retail inventory robot

MASTER THESIS

Jordi Soler Busquets

Msc. Automatic Control and Robotics candidate, UPC

jordi.solerbusquets@gmail.com

Supervisors

Rafael Pous
Universitat Pompeu Fabra

Cecilio Angulo
Universitat Politècnica de Catalunya

Victor Casamayor
PhD student
UPF

Marc Morenza
PhD student
UPF

SEPTEMBER 2016

Abstract

Many mobile robotic systems produce a 3D model of their environment by sweeping it with their RGBD (color and depth) cameras. In the case of the Advanrobot, an automatic inventory wheeled robot for retail stores, such ability opens the door to a range of new possibilities such as building a virtual store or helping retailers in measuring the impact of the placement of products, furniture and their combination in the sales.

This work presents a solution to enable Advanrobot for building 3D models from its RGBD camera measures. To do so, external tools have been used to handle navigation, localization, mapping and point cloud combination so that the main development is focused on 3D exploration to ensure 3D model completeness. As navigation is limited to 2 dimensions, the ground projection of the camera point clouds is used as input for a frontier based exploration approach consisting in a flexible framework able to extensively tune exploration decisions such as the evaluation of frontiers, when should the robot decide a new navigation goal or how to choose a proper goal when a frontier has already been chosen. Also, some additional tools have been developed as spin-offs of the work such as a hole detector, a point cloud trimmer or a camera calibrator.

The approach has been proved to be useful for automatic 3D mapping of the environment. Exploration based mapping lead to similar results than teleoperated mapping. Each related tool also solved its corresponding problem.

The results of this work may apply to any wheeled robot with a fixed RGBD camera.

Contents

1 Introduction

One important application of RGBD cameras is to build a 3D model of the environment as the camera moves through it. This project aims at developing a tool to enable the Advanrobot robot of Keonn Technologies do 3D models of the space atunomously. Robots having a static RGBD camera have the sensor focus determined by the robot position and hence from the desire of environment model completeness arises the need of a thoughtful robot motion in order to capture all the needed details. Taking into account that the Advanrobot already has a navigation system, in order to achieve a desired motion the developed work has its focus on a proper choice for the robot goals which make it sweep the environment as efficently and completely as posible.

In order to solve some issues related to working with point cloud data, many other functionalities have been added such as hole detection, point cloud calibration, point cloud trimming and more.

1.1 Motivation

Having a 3D model of the environment opens a new range of possibilities, specially when combined with product localisation. Such possibilities include a virtual shop, analyse the impact of product location or navigation and localisation advantatges for the robot itself. The new developed features pretend to solve problems found at both developing and testing stage. Thus, the need of point cloud managing tools arise from the real sensor data observed when testing.

1.2 Advanrobot

The Advanrobot is an inventory robot aimed for RFID based automatic inventory in retail stores.

Figure 1: Advanrobot inventory robot

It is a ground robot with two motor wheels and three omnidirectional wheels that enable free ground movements. Its equipped with a LIDAR Hokuyo sensor and an Xtion Pro RGBD camera.

1.3 Software framework

1.3.1 ROS environment

ROS Overview

All the robot software is integrated in ROS (Robot Operating System). ROS adds a layer of software providing a communication infrastructure and also provides useful tools to handle common robotics procedures. ROS support to programming languages is mainly focused on C++ and Python. The code developed during this work has mainly been developed in C++. The ROS version used in this work is ROS Indigo which runs on Ubuntu 14.04.

In a ROS environment the code is split in different units called nodes that are able to interact with one another. The communication infrastructure is based on the facilities:

- **Messages**. Messages are data structures based on a publication-subscription scheme. Each message has a *type* that has to be defined in a file. No custom messages have been defined for this project since many standard types are defined in the ROS distribution.

  One node that wants to share a message with the network, *publishes* it in a certain channel which is called *topic*. Then, that message is accessible to any node that is *subscribed* to that topic. As many nodes can be subscribed to the same topic, the communication is said to be one-to-many.

  This communication facility has extensively been used in the development of this project.

- **Services**. Services share data between two nodes in a request-response scheme, so they allow synchronous communications. In this project no services have been used.

- **Actions**. As in the case of services, actions follow a request-response scheme, but they allow reporting the service status during their execution. In this work an action have been used to communicate with the *move_base* node of the navigation stack. In this case, a navigation goal is send and the the navigation state is reported when the goal is no longer active.

- **Parameter Server**. The parameter server allows setting and getting key-value pairs from nodes, files and command-line. This facility plays a central role in the development of an exploration framework, section Exploration of a 3D environment using RGBD cameras.

ROS tools and their application in this work

Some nodes, message types and tools which are delivered with the ROS distribution need to be explained in order to understand the development in this project.

The ROS navigation stack [3] is used and fully configured to work with the robot. It provides a navigation system that handles the motion of the robot from point to point. The main components of the navigation system are:

- **The *move_base* node**. This ROS node plays a central part in the navigation stack. It provides the interface with the user code through an action client. Some utilities of the navigation stack, such as the costmaps and planners commented below, are plugins of the *move_base* node.

- **Global and local costmaps**. In navigation, the need of knowing if certain regions are occupied give rise to the building of costmaps. In the case of Advanrobot, the navigation stack is configured such that it uses a global costmap, that contains all the information available regarding the occupancy of the navigation space, and a local costmap which stores local information based on the recent sensor readings. Both costmaps publish a message of type */nav_msgs/OccupancyGrid* which builds a grid corresponding to the robot navigation space where each cell is assigned a probability to be occupied. In the developed strategy, this information is used to check whether a point in the map is occupied. Also the configuration of the costmaps has been changed to study the viability of adding an extra camera.

- **Global and local planners**. When a new navigation goal is received by the *move_base*, a path from the robot to the goal is planned so that it avoids the obstacles in the costmaps. This global planning process is done using Dijkstra's shortest path algorithm [1]. To follow this global path a local one is planned by the local planner. This local planner uses the Trajectory Rollout algorithm [5]. This planners are suggested to be used in future development of this work, see section Future work proposals.

Using the navigation stack, to make the robot go to a certain goal, it is enough to send the goal's coordinates to the *move_base*.

Aside from the navigation stack, some ROS features related with this project need to be discussed for clarity.

The ROS *tf* package keeps track of different coordinate frames over time, making it simple to handle data from different sources. Trees of frames are stored so that any frame can be related to any other one, as long as they are part of the same network. For connectivity, it is desirable

that all frames lie in the same network, as is the case in this project. It has extensively used in tasks such as translating and rotating sensor data or computations involving time or durations.

The depth data stream coming out of the RGBD camera ROS driver is available through a published message of type *sensor_msgs/PointCloud2*. This message type represents a cloud of points attached to a frame at a certain time. The data coming out of a RGBD camera can be thought as 4 matrices, updated periodically. Three of these matrix correspond to the levels of red, blue and green colours, as in standard color image. The extra matrix encodes the depth measure for each pixel. Since the message is attached to a frame, the point coordinates corresponding to each pixel in whichever frame can be computed.

Simulation

For simulation purposes, the robot is described by URDFs (Universal Robot Description Files) which among many other parameters, contain the RGBD camera position. As the introduction of a second camera is under study, the robot model has been tuned by adding an extra RGBD camera and also, by trying different configurations (positions and orientations) for the cameras.

With a robot model, the Gazebo simulator enables simulation of the robot in a certain environment. The Gazebo simulator can be integrated in ROS to perform the simulation of the physical robot-environment system. Gazebo simulations have been extensively used as first step testing.

For visualisation purposes the RVIZ application has been used.

1.3.2 The Point Cloud Library (PCL)

The Point Cloud Library is an open community-driven project [12] for point cloud processing. It provides tools for filtering, segmenting, registering and feature extraction from point clouds. This library is wrapped in ROS by *pcl_ros* library. The point cloud ROS message types are compatible with the library types.

1.3.3 The RGBD camera drivers

The main interface with the hardware is done by OpenNI2. OpenNI2 is an opensource project derived from OpenNi (*Open Natural Interaction*) which aims at providing a natural using interface for natural interaction devices (RGBD cameras). This software layer is wraped in ROS by the package *openni_camera*.

1.4 State of the art

The use of RGBD cameras is extensive in robotics with applications ranging from visual SLAM [11] to navigation and obstacle avoidance [9, 10]. Also, there exist a number of mature techniques to tackle the robot exploration problem. One family of such is the frontier-based exploration which has long been exploited since its introduction in [15]. In spite of both facts there is not a clear path to follow for the RGBD camera exploration problem in the sense of 3D model building. In [2] this problem is discussed for the case of a RGBD camera mounted on a mobile platform where 3D capabilities are exploited. In this case the techniques described are not suited for the case of a fixed sensor with 2D mobility, where looking for unexplored 3D spaces provides no additional useful information.

2 Exploration of a 3D environment using RGBD cameras

Since there is a lack of both an implementation and a well established procedure for 3D environment modelling a new approach has been developed. The basic idea of the approach is to collapse the point cloud stream coming out from the sensors into the ground, and then implement a 2D frontier-based exploration that considers the special features of the sensor data.

In the collapsing process the height information is lost. A 2D based approach has to be build upon the assumption that there exist a 2D navigation goal that makes the frontier point visible for the RGBD system. To meet that requirement in most cases one could use a mobile platform to move the sensor which would introduce greater flexibility while increasing the cost of the sensing system and potentially decrease the accuracy in the camera position and orientation estimations. Another more robust approach is to use more RGBD sensors, thus increasing the visibility. In this work this second approach is studied in simulation as it fits the needs of the robot.

For the ground projection process the external ROS package *rtab_map* has been used. This package is a wraper of Real-Time Appearence-Based Mapping approach based on the work in [6, 7, 8]. It provides both a ground projection of the point cloud data and builds an environment map that can be build in several sessions.

The remaining work is the development of a 2D exploration based on the projected data. To do so a full exploration framework has been developed that allows testing different policies.

2.1 RGBD camera exploration

The developed camera exploration framework is implemented as a ROS node named *cam_exploration*. The top process flow follows the following behaviour.

1. Initialise node, setup subscribers and publishers.
2. For each new point cloud ground projection:
    a) Compute the frontiers and update a frontier map.
    b) Check whether to send a 2D navigation goal.
    c) If so
        i. Select the best frontier according to some criteria.
        ii. Select a proper point of such frontier.

As can be seen, the node is intended to work permanently until system shutdown.

2.1.1 Frontier map

A collection of 2D frontier representations are stored in memory. For each one, basic information is stored.

- **Map cells**: Map cells corresponding to the frontier.
- **Identifier**: The frontiers need to be uniquely named.
- **Center point**: Useful for frontier evaluation.

2.1.2 Frontiers detection

Incoming point cloud ground projections are processed in order to build a 2D map from which to detect new frontiers. The frontier detection is made using the two-pass connected-components labeling algorithm [14]

2.2 Sending a new 2D navidation goal

As pointed out in [13], in the context of frontier-based exploration, one could benefit from map updates while in the way to a current goal. That can avoid the robot to go to an outdated goal, that can also serve to head to a more valuable goal if the robot suddenly realizes that the current goal is no longer the most valuable.

To decide whether to send a new goal, a set of replanning conditions, each one corresponding to a different criterion, have been used. This replanning condition can be combined between them, so that their combination is what actually determines whether to send a new navigation goal. The combination method is an *OR* operation, so if any condition is met, the robot should send a new goal.

2.2.1 Too much time near the goal

It votes for replanning if the robot has spent some time near its current goal and also if its properly oriented with the goal. This condition can be expressed as.

$$(t_{ng} > t_{th}) \;\&\; (y_{rg} > y_{th}) \tag{1}$$

Where t_{ng} is the time that the robot has spent within a certain circle centred in the current goal with radius r_{th}. The difference between the actual robot orientation and the desired orientation is y_{rg}. Values t_{th} and y_{th} are the thresholds in time and orientation respectively and

$$y_{rg} = \angle(\vec{r}, \vec{g}) \tag{2}$$

With $\vec{r}$ and $\vec{g}$ the robot and goal orientations.

2.2.2 The goal is not close to any frontier

When the robot approaches the goal, new sensor data arrives, some times involving point cloud projections where the frontier that originated the goal was. It is a usual case since the RGBD camera of the Advanrobot aims in the direction that it moves. In such scenario the original goal frontier is modified or completely erased, therefore an update in such frontier is desirable.

This replanning condition is activated when some cell in the neighbourhood of the goal cell corresponds to a frontier cell. The neighbours are are obtained with the algorithm 5

Algorithm 1 Get neighbours

input: Center cell c_{center}, depth d;
$\mathcal{C} = \emptyset$
$c_{up} = c_{down} = c_{right} = c_{left} = c_{center}$
for i in 0 to d **do**
    $c_{up} = \text{up}(c_{up})$
    $c_{down} = \text{down}(c_{down})$
    $c_{right} = \text{right}(c_{right})$
    $c_{left} = \text{left}(c_{left})$
    $\mathcal{C} \leftarrow c_{up}, c_{down}, c_{right}, c_{left}$
    for j in 0 to i **do**
        $c_{ne} = \text{downright}(c_{ne})$
        $c_{se} = \text{downleft}(c_{se})$
        $c_{sw} = \text{upleft}(c_{sw})$
        $c_{nw} = \text{upright}(c_{nw})$
        $\mathcal{C} \leftarrow c_{ne}, c_{se}, c_{sw}, c_{nw}$
return $\mathcal{C}$;

2.3 Evaluating frontiers

The main policy to be studied in frontier-based exploration is the choice of the goal frontier among the complete set of frontiers. To achieve that, a cost function is usually defined taking into account some criteria. As before, some frontier evaluation methods are defined which can be combined. Thus, for a set up with n frontier evaluation criterion that evaluate each frontier, the best frontier is the one that maximizes

$$\arg\max_{f} \sum_{i=0}^{n} w_i \cdot h_i(f) \qquad \text{with } f \in \mathcal{F} \tag{3}$$

Being $\mathcal{F}$ the set of current frontiers and h_i and w_i the i-th frontier evaluation function and its weight.

2.3.1 Maximum size

It assigns a value to a frontier equal to the number of map cells that correspond to the frontiers.

$$h_{max\ size}(f) = \text{size}(f) \tag{4}$$

Biasing the exploration towards larger frontiers helps focusing on the main unexplored parts before caring about some unexplored details. For instance, it may favour the exploration of a whole room before exploring another one.

2.3.2 Minimum euclidean distance

It favours frontiers with a reference point closer to the robot position. Both points are considered to lie in the XY plane, and hence, the distance measure for a given frontier f is

$$d_{euc}(f) = \sqrt{(r_x - f_x)^2 + (r_y - f_y)^2} \tag{5}$$

Being r_x and r_y the robot position coordinates and f_x and f_y the frontier's reference point coordinates. The point corresponding to the frontier center cell is taken as the reference point.

The actual value given to a frontier is a function of the distance measure.

$$h_{min.euc.dist.}(f) = e^{-\frac{d_{euc}(f)}{\lambda}} \tag{6}$$

Where λ is a dispersion factor determining the locality of the evaluation criterion. It should be noticed that there is no parameter pre-multiplying the right hand side of the equation 6 as $h_{min.euc.dist.}$ is supposed to be multiplied by the tunable weight $w_{min.euc.dist.}$.

2.3.3 Minimum $\mathcal{A}$* distance

The $\mathcal{A}$* algorithm is intended to find a minimal cost path from a start point to a goal point in a grid. This optimal path is measured and used as a distance measure, $d_{\mathcal{A}*}$. Then, an equation analogous to 6 is used to obtain the frontier value.

An integrated $\mathcal{A}$* implementation has been developed using the following algorithm.

Algorithm 2 $\mathcal{A}^*$ algorithm

input: Start node n_s, final node n_f;
$\mathcal{O} = \emptyset$
$\mathcal{C} = \emptyset$
$\mathcal{O} \leftarrow n_s$
while True **do**
    $n_c = \arg\min_n f_{cost}(n)$ with $n \in O$
    $\mathcal{O}$ removes n_c
    $\mathcal{C} \leftarrow n_c$
    if $n_c = n_f$ **then**
        return $traceback(n_c)$
    for all $n_n \in neighbours(n_c)$ **do**
        if n_n not traversable **or** $n_n \in \mathcal{C}$ **then**
            Skip until next n_n
        if $newCost(n_n) < get(cost, n_n)$ **or** $n_n \notin \mathcal{O}$ **then**
            $set(cost, n_n) \leftarrow newCost(n_n)$
            $set(parent, n_n) \leftarrow n_c$
            if $n_n \notin \mathcal{O}$ **then**
                $\mathcal{O} \leftarrow n_n$

Where $neighbours(n_c)$ are the set of neighbours of n_c considering 8-connectivity, a node n is said to be traversable if the corresponding cell is not occupied and the $traceback(n)$ function returns the set of cells from n to the starting node by jumping from each node to its parent as in algorithm 3.

Algorithm 3 Traceback

input: Node to be handled n_i, first node n_s;
$\mathcal{P} \leftarrow \emptyset$
$n_c = n_i$
while $n_c \neq n_s$ **do**
    $\mathcal{P} \leftarrow n_c$
    $n_c = get(parent, n_c)$
$\mathcal{P} \leftarrow n_c$
return $\mathcal{P}$

Finally, the evaluation function analogous to 6 that provides the frontier value is

$$h_{min.\mathcal{A}*}(f) = e^{-\frac{d_{\mathcal{A}*}(f)}{\lambda}} \tag{7}$$

2.4 Choosing a proper goal

Even when a frontier is selected as exploration target the decision of a proper 2D navigation goal is not straightforward. In a scenario where frontiers where build from horizontal laser rangefinder data one could simply chose a the frontier middle point and go directly there. In the case of the RGBD camera exploration, it is needed that the goal is positioned so that when the robot is at the goal its camera is facing the frontier and, thus, erases it. Otherwise the robot could find itself stuck in a deadlock where the goal is produced by a frontier which persists when the robot is in the goal, leading to an endless loop.

The procedure to obtain a valid goal is based on a frontier point. In testing the frontier's middle point has been used, but any other point of interest from the frontier could be used. The goal is then chosen to be at a certain distance of such point and the orientation of the goal should aim at the frontier point. The goal point needs to be a valid point and the frontier point should be at sight from the goal (no obstacles in between).

This behaviour is defined in the algorithm 4 and exemplified in figure 2. It should be noticed that algorithm 2 is a general algorithm to which one should add an exit condition for the case where no valid point is found in the circumference centred in the point p_v after some iterations. In practice this condition is met when a certain number of different resolution spins have been completed.

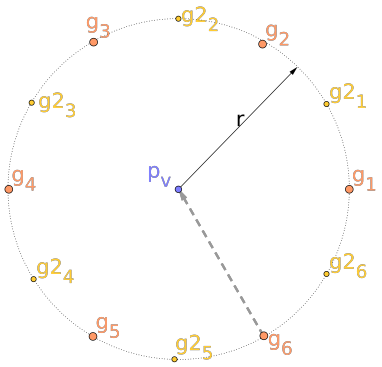


Figure 2: Example of goal candidates for a frontier point p_f with initial divider $d_i = 3$. The orange points indicate the first set of points, and the yellow the second. Further candidates are not shown for clarity. The grey dotted arrow indicates the line of sight to be checked for the validity of goal candidate g_6. The robot position p_r is supposed to be on the right.

Algorithm 4 Aim at a frontier point

input: Frontier's point of interest p_f, robot position p_r, distance goal-frontier point r, initial divider d_i;
g: goal candidate position
$\vec{v}_\alpha$: Vector with length r and angle α
$\alpha = \angle p_f, p_r$
$g = p_f - \vec{v}_\alpha$
$d = d_i$
while g is not free or has not p_f at sight **do**
    if A whole turn has completed with sampling d **then**
        $d = d_{prev}/2$
    $\alpha = \alpha_{prev} + \pi/d$
    $g = p_f - \vec{v}_\alpha$
return g

3 2D navigation and developed tools

Navigation using RGBD information has been proven to present some differences when compared with LIDAR-based approaches. The differences are mostly related with accuracy and maximum range ofthe sensors as RGBD cameras in general provide poorer results in both performance measures.

There are some problems to be tackled in 2D exploration regarding the use of RGBD cameras, as well as new features to be developed. In this work both, problem solutions and new features are developed. Below there are the main tools developed in this project.

3.1 Hole detection

One of the main concerns from Advanrobot users is its ability to handle holes, in a general sense, including cliff-like scenarios like descending stairs. Therefore, a hole detection software tool has been build so that holes are detected and labelled as obstacles.

From the sensor data, hole information can be detected through data points that are below the robot's base or by NaNs (Not a Number) data points. This latter case, in practice, corresponds not only to out-of-range measures but also to misread measures. That, along with the fact that NaNs provide no information regarding where the corresponding obstacle should be located, has led to the development of two different approaches to handle points under the robot base and NaNs.

3.1.1 Sensed points below the robot base

For each sensed cloud point below the robot base, P, an obstacle point, O, is created as shown in figure 3. Locating O as a function of P allow to properly deduce the obstacle location. As the hole should be located where the ray would hit the ground if there was no hole,

$$\begin{pmatrix} O_x \\ O_y \end{pmatrix} = \frac{h}{h - P_z} \cdot \begin{pmatrix} P_x \\ P_y \end{pmatrix} \tag{8}$$

Being h the height of the robot and (P_x, P_y) and (O_x, O_y) the coordinates of the sensed point and the obstacle point. Note that the image 3 shows a simplified version where the point is in the ZX point but in general one should consider the general case where this problem can be rotated along the Z axis. Another simplification in the figure is that the points are considered to be below the robot base if they are under a negative threshold: $P_z < -z_{th}$ being z_{th} a positive value. The remaining obstacle point coordinate is set such that the point is considered as an

obstacle.

$$O_z = h_{obs} \tag{9}$$

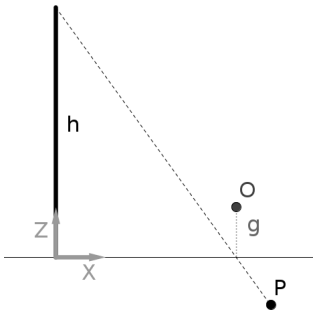


Figure 3: Obstacle declaration from a point below the robot base. The solid vertical line represents the Advanrobot. Point P is a sensed point below the robot base ($P_z < 0$). The point O is the obstacle point corresponding to P.

When all points are processed a point cloud with the created obstacle points is published so that the obstacles are taken into account. These resulting obstacles are to be considered as lasting holes that should never be erased by other sensor data that cannot sense the holes.

3.1.2 NaN data points

In the case of holes where the bottom is not found by the camera (stairwells, cliff-like scenarios, etc.) NaN values are returned corresponding to the hole points. In this case, the position of the obstacle corresponding to a NaN cannot be deduced from the point coordinates. In this case, the proposed solution includes two stages:

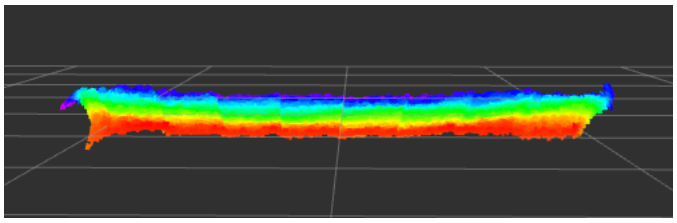
- **Structure recognition.** When the robot camera is facing the ground, a point cloud is stored to be taken as reference.
- **Hole detection.** The reference ground projection is loaded. When a new point cloud arrives it is point-wise iterated so that an obstacle point is generated for each NaN value. Such obstacle point is located in the XY plane by giving it the same X and Y coordinates as the point from the reference projection with the same indexing.

The Z coordinate of the obstacle point is set to an arbitrary obstacle height h_{obs} as in the previous case 9.

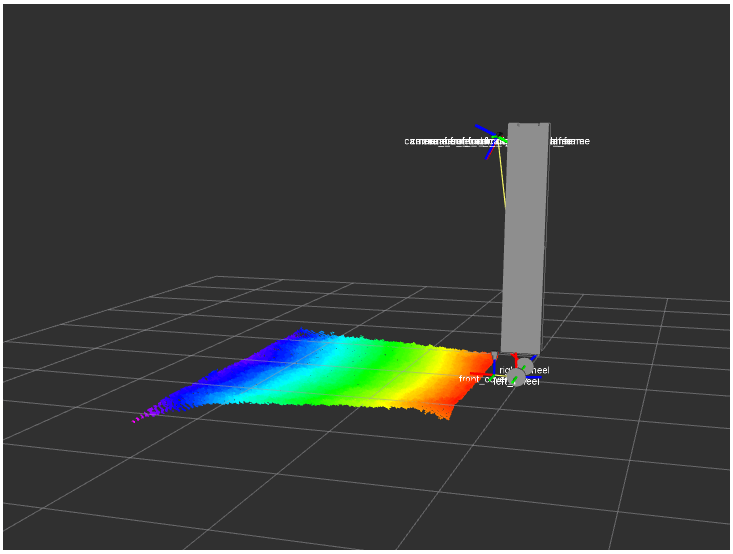
This hole detection procedure is performed with the same iteration that in the previous case where obstacle points are sensed as valid spatial points. Another different point cloud is published in this case representing the NaNs information. This cloud does not reliably represent holes as in the previous case, but it also includes points that are misread for whatever reason.

3.2 Point Cloud calibration

The mounted RGBD camera provides readings that suffer slight deformations even after driver calibration, see figure 4. To correct the point cloud data a plane fitting procedure is performed, which consists in two stages.



(a) Back view.



(b) Point cloud with the robot.

Figure 4: Point cloud ground projection from the top RGBD camera.

3.2.1 Calibration stage

In a first stage, the robot should be located in a planar surface such that its RGBD camera is facing the ground, with no obstacles. At this point a reference point cloud is considered and a plane is fitted to that point cloud that should correspond with the ground. The algorithm used for fitting the plane is RANSAC (RANdom SAmple Consensus) [4].

Once the plane model parameters are obtained in the form

$$Ax + By + Cz + D = 0 \tag{10}$$

for each point in the point cloud (in the camera frame) P_i^{cam} a depth correction, z_{off}, is stored

Algorithm 5 RANSAC

input: set of observed data points *data*
    threshold of distance to model to consider a point an inlier d_{th}
    number of iterations i_{max}
    model that can be fitted to *data* with n parameters *model*
for i in 0 to i_{max} **do**
    $R_n = n$ points randomly selected from *data*
    $inliers = R_n$
    model = model fitted from R_n
    for point p in *data* **do**
        if distance from p to the model *model* $< d_{th}$ **then**
            $inliers \leftarrow p$
    model = model fitted from *inliers*
    if *model* is better than *best* (according to some performance measure) **then**
        $best = model$
return *best*;

in a file. With the plane model parameters it is computed as follows:

$$z_{i,plane} = \frac{1}{C} \cdot (-D - AP_{i,x}^{cam} - BP_{i,y}^{cam}) \tag{11}$$

$$z_{i,off} = z_{i,plane} - P_{i,z}^{cam} \tag{12}$$

3.2.2 Correction stage

Once the calibration has been made, to correct a point cloud it it enough to add the proper offset to each point in each incoming cloud. Hence, we have

$$P_{i,corrected} = P_{i,sensor} + z_{i,off} \tag{13}$$

to be applied to all points.

3.3 Point cloud trimmer

One possibility for filtering point cloud data is to manually trim point clouds so a certain region of them is not used for further computations. This may come at handy when observed point cloud data has too noisy regions, when all points of a region of the point cloud are always NaNs, or when the region of interest for further point cloud treatment is narrower.

The introduction of this tool is motivated by the imperfections of the RGBD camera mounted in the Advanrobot. It presents a ban of NaN points in the left border and also very noisy parts in the corners, see figure 5.

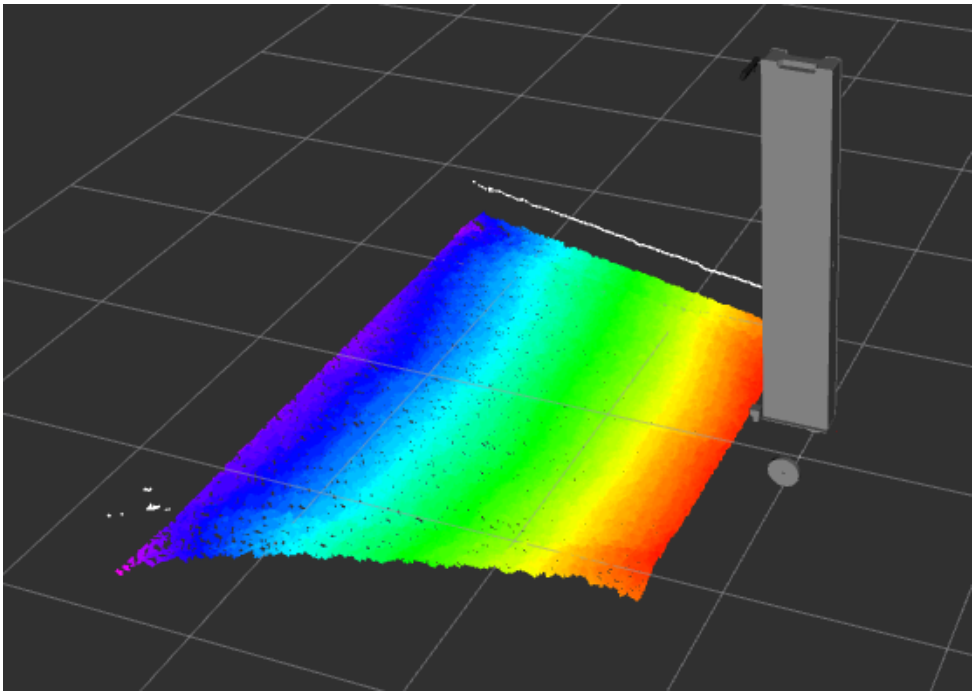


Figure 5: Point cloud ground projection. The white points indicate either NaNs or points below a certain threshold level. This points are a result of the *hole_detection* node.

3.3.1 Point cloud structure

One way of selecting the region of the point cloud to be trimmed is by conditioning the validity of a point by its x and y coordinates in the corresponding depth image. This allows the recognition of corners and sides which can be used when selecting a trimming region.

Conversely, the information of the point cloud data being published is only accessible by a single index and no information about the point cloud shape is provided. For this reason, an algorithm has been developed in order to deduce the point cloud shape from the sensed data itself and hence make the x and y point reference possible. This algorithm is shown in algorithm 6.

Once *width* and *height* are known, the conversion from single index and double indexing and vice-versa is straightforward. These functionalities are wrapped in the *pc_structure* library.

Algorithm 6 Detect shape

input: point cloud points, *points*
        threshold of distance between consecutive points, d_{th}
P_i : i-th point of *points*
$W = \emptyset$
$i = 1$
while P_i exists **do**
    if $|P_i - P_{i-1}| > d_{th}$ **then**
        $W \leftarrow |P_i - P_{i-1}|$
    increment i
width = most frequent value in W
height = *width*/i
return *width* and *height*

3.3.2 Point cloud trimmer node

The ROS node in charge of trimming the point cloud is *pc_trimmer*. This node is subscribed to a point cloud source and publishes a trimmed version of the incoming stream. It uses the *pc_structure* library allowing the definition of a trimming function as a point-wise function depending on x, y, and the *width* and *height* of the point cloud. Figure 6 shows a custom trimming function.

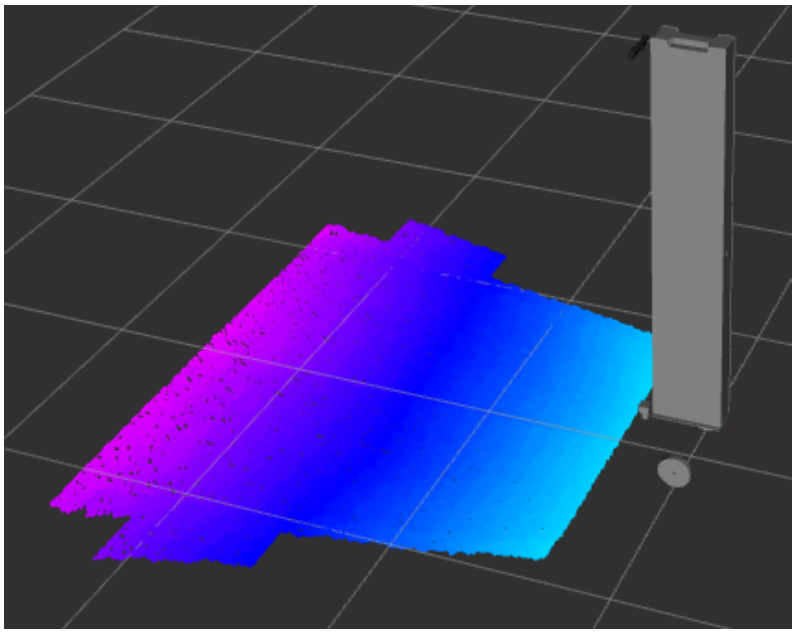


Figure 6: Trimmed point cloud with a T-shape.

3.4 Other tools

3.4.1 Range publisher

Regarding the consideration of adding a second RGBD camera to the Advanrobot, some tests have been made. For that purpose, the *range_publisher* node has been created to publish messages corresponding to the field of view of the tested camera system.

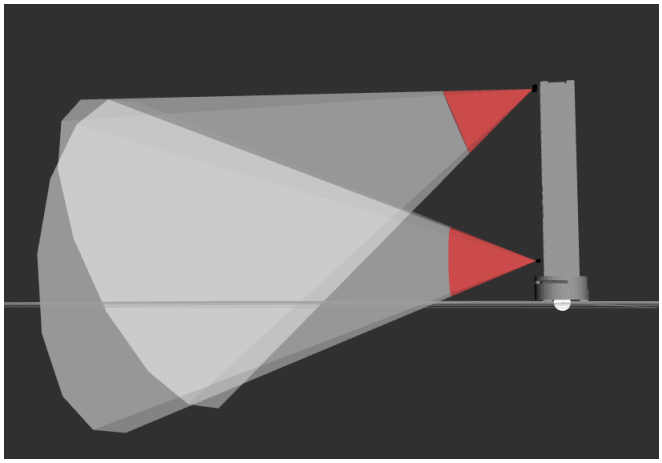


Figure 7: Example of published sensor ranges for testing RGBD cameras setup. Both top and bottom cameras are simulated Asus Xtion models. The red zone is not visible by the cameras due to their minimum distance.

3.4.2 Cloud merge

In case of adding a second camera, two different point clouds would be published. For some applications it may be interesting to treat point clouds streamed from both cameras, as a hole. To do so, the *cloud_merge* node has been developed to join both sources. It subscribes to both single camera point clouds and published a merged version of it in another topic. The point cloud merging task is forwarded to the PCL library.

4 Code summary

4.1 RGBD camera exploration

The general data flow starts with the RGBD readings from the sensor driver which are published as ROS point cloud messages. This messages are used by *rtabmap* node which builds a 3D model from single point clouds and also publishes the projection of this model in the ground as a map, allowing to differentiate the unexplored regions to the ones corresponding to the 3D model projections. At this point, the *cam_exploration* node uses this published map for exploration.

All the map-related information is handled by the *map_server* library. The node also provides visual information of its state using markers, which are handled by the *marker_publisher* library. To keep track of the robot location and handle the interaction with the *move_base* node, the *robot_motion* library is used.

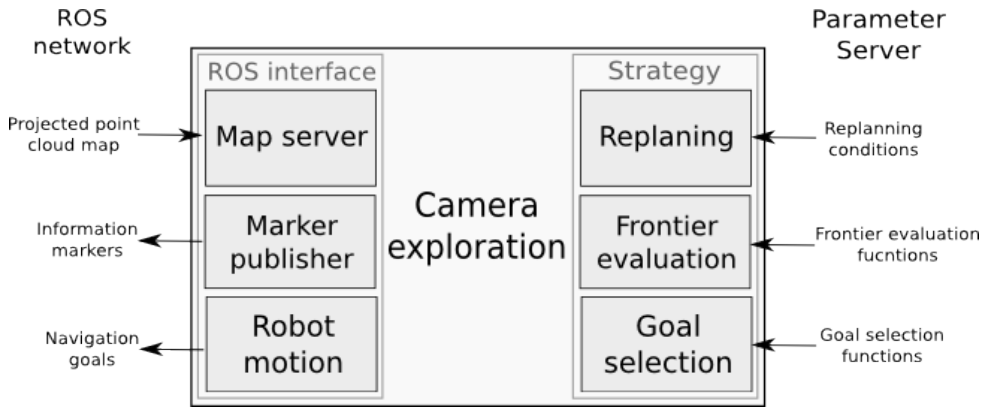


Figure 8: Main *cam_exploration* structure with its main libraries.

The main feature of this framework is its modularity. As 3D mapping is a new use case in development stage, the needs for versatility in testing different strategies is a key requirement. The main strategic decisions to be taken can be extensively configured through the ROS parameter server.

4.1.1 ROS API

The integration with ROS can be described with the node main subscriptions, publications and parameters.

Subscribed topics

- **/proj_map** (*nav_msgs/OccupancyGrid*)

Incoming map from *rtabmap* consisting in 3D camera point cloud projections.

Published topics

- **/goal_padding** (*visualization_msgs/Marker*)

  Region considered as the goal neighbourhood for robot-goal proximity purposes.

- **/goal_frontier** (*visualization_msgs/Marker*)

  Target frontier.

- **/goal_marker** (*visualization_msgs/Marker*)

  Selected goal point.

Parameters

- **~/frontier_value/functions** (*list(string)*, default: [])

  List of frontier evaluation functions to be used. Possible values are *max_size*, *min_euclidian_distance* and *min_astar_distance*

- **~/<function>/weight** (*double*, default: 1.0)

  Value used to weight the function <function>.

- **~/min_euclidian_distance/dispersion** (*double*, default: 1.0)

  Degree of locality of the function min_euclidian_distance.

- **~/min_astar_distance/dispersion** (*double*, default: 1.0)

  Degree of locality of the function min_astar_distance.

- **~/minimum_frontier_size** (*int*, default: 15)

  Minimum number of cells of a frontier to be considered a target candidate.

- **~/goal_selector/type** (*string*, default: *"mid_point"*)

  Way of choosing one of the target frontier points for target point. Only *mid_point* is implemented

- **~/distance_to_goal** (*double*, default: 1.0)

  Distance between the actual 2D navigation goal target frontier point. Should be close to the usual distance from the robot footprint to the nearest 3D camera point cloud projection point.

- **~/replaning/conditions** (*list(string)*, default: [])

  List of replanning conditions to be applied. Possible options are *not_moving*, *too_much_time_near_goal* and *isolated_goal*.

- **~/too_much_time_near_goal/time_threshold** (*double*, default: 0.3)

  Maximum time in seconds allowed for the robot to be near a goal in *too_much_time_near_goal* replanning condition.

- **~/too_much_time_near_goal/distance_threshold** (*double*, default: 0.5)

  Minimum distance from the goal at which the robot is considered to be near it in *too_much_time_near_goal* replanning condition.

- **~/too_much_time_near_goal/orientation_threshold** (*double*, default: 0.5)

  Maximum orientation difference between the one of the robot and the one of the goal, to allow replanning in *too_much_time_near_goal* replanning condition.

- **~/isolated_goal/depth** (*int*, default: 5)

  Minimum rectangular distance from the goal to its nearest frontier allowed without replanning in *isolated_goal* replanning condition.

4.2 Hole detection

The code related with hole detection is wrapped in the *hole_detector* ROS node. This node is part of the *cam_scripts* package.

4.2.1 ROS API

The main ROS communications of the node *hole_detector* are described below.

Subscribed topics

- **/tf_static** (*tf2_msgs/TFMessage*)
  Static frame tree from the robot URDF. It is used to transform the incoming point clouds from the camera frame to the robot footprint one.

- **/cloud_in** (*sensor_msgs/PointCloud2*)
  The point cloud from which the holes are to be obtained. Usually the raw sensor point cloud.

Published topics

- **/hole_detector/holes** (*sensor_msgs/PointCloud2*)
  Point cloud containing the detected holes. Holes are represented by regions with a certain height and a position in the XY corresponding to the deduced position of the holes.

- **/hole_detector/nans** (*sensor_msgs/PointCloud2*) Point cloud containing regions of NaNs in the source cloud. Such points are located where they would be if the corresponding beam reached the ground and returned a proper value. Their height is parameterized to allow obstacle detection.

Parameters

- **/hole_detector/obstacle_height** (*double*, 0.2)
  Height (from the base footprint) at which the obstacle points should be placed. It should be related with the obstacle height parameter in the navigation costmap plugins.

- **/hole_detector/obstacle_distance** (*double*, 3.0)
  Maximum distance allowed for a point to be handled.

- **/hole_detector/minimum_height** (*double*, -0.01)
  Maximum height for a point to be considered part of a hole.

- **/hole_detector/camera_link** (*str*, *"/camera_link"*)
  Source frame for point cloud messages.

- **/hole_detector/projection_in** (*str*, *""*)
  Name of the file from which the reference trajectory should be loaded for handling NaNs.

4.3 Point cloud calibration

The code related to point cloud calibration is wrapped in the ROS package *pc_calibrator*. The package provides two ROS nodes. The *pc_calibration* node is intended to be used in calibration stage, with the RGBD camera facing the ground, with no obstacles. The *pc_corrector* node uses a previously generated calibration to correct point clouds online. The correction is stored in a *csv*-like file.

The main functionality of this package comes from the developed library *pc_calibration* which could be used to patch directly the source of point clouds instead of using the subscription-publication scheme of ROS (e.g. using the provided nodes).

4.3.1 Node pc_calibrator ROS API

Subscribed topics

- **/cloud_in** (*sensor_msgs/PointCloud2*)
  Point cloud to be calibrated.

Published topics

- **~/plane_coefficients** (*pcl_msgs/ModelCoefficients*)
  Coefficients of the fitted plane.

- **~/poly_pub** (*geometry_msgs/PolygonStamped*)
  Square embedded in the fitted plane for visualisation purposes.

Parameters

- **~/file_name** (*str, "Calibration.csv"*)
  File to which the calibration is to be written. Paths are relative to the package *pc_calibration* unless full path is specified with preceding "/".

4.3.2 Node pc_corrector ROS API

Subscribed topics

- **/cloud_in** (*sensor_msgs/PointCloud2*)
  Source point cloud to be corrected.

Published topics

- **~/corrected_cloud** (*sensor_msgs/PointCloud2*)
  Corrected version of the source point cloud.

Parameters

- **~/file_name** (*str, "Calibration.csv"*)
  File from which the calibration is to be read. Paths are relative to the package *pc_calibration* unless full path is specified with preceding "/".

4.4 Point cloud trimmer

The main developed ROS tool for trimming point clouds is the *pc_trimmer* node. As in the case of *pc_calibration*, the main functionality comes from a developed library, in this case also called *pc_trimmer* for allowing greater portability. Both node and library are part of the *cam_scripts* package.

4.4.1 ROS API

The ROS point cloud trimming interface is the *pc_trimmer* node which connections with the ROS network are the following.

Subscribed topics

- **/cloud_in** (*sensor_msgs/PointCloud2*)
  Point clouds to be trimmed.

Published topics

- **/cloud_out** (*sensor_msgs/PointCloud2*)
  Trimmed point clouds.

Parameters

- **~/offset_left** (*float*, 0.0)
  Percentage of the point cloud width to be trimmed on the left.
- **~/offset_right** (*float*, 0.0)
  Percentage of the point cloud width to be trimmed on the right.
- **~/offset_top** (*float*, 0.0)
  Percentage of the point cloud height to be trimmed on the top.
- **~/offset_bottom** (*float*, 0.0)
  Percentage of the point cloud height to be trimmed on the bottom.

5 Results

5.1 RGBD camera exploration

The RGBD camera exploration framework has been tested in both simulation and real robot with slightly different setup.

5.1.1 Simulation

The simulation setup considers two different RGBD cameras, FIGURE. This allows the robot to capture features in a wider range of heights and boosts the resolution. The used environment for testing is a model of the WillowGarage® offices, FIGURE. This big environment has been used to explore parts of it when testing individual features of the framework.

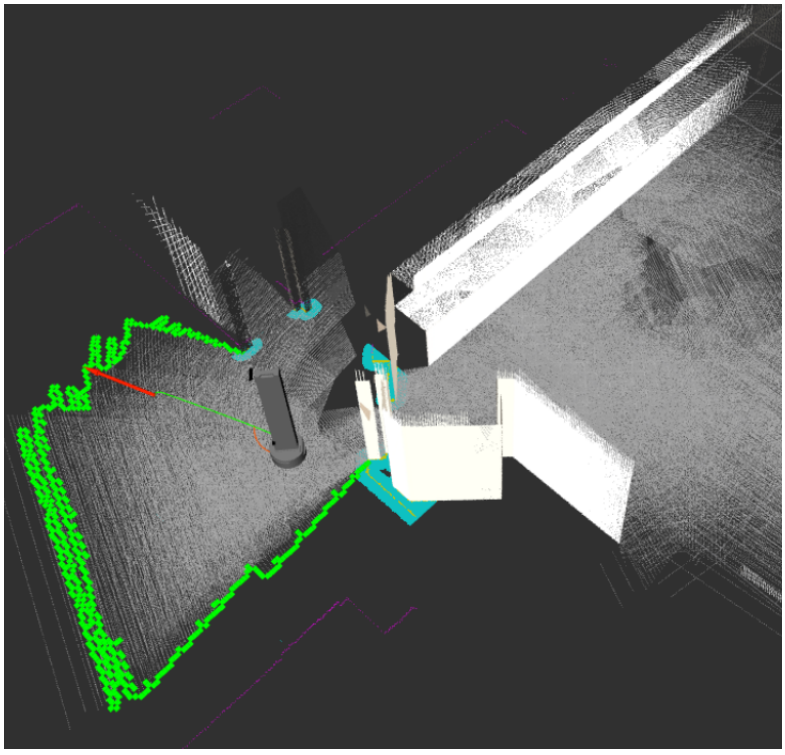


Figure 9: 3D model building with the Gazebo simulator. The green dots indicate the target frontier, the red arrow correspond to the navigation goal pose, the green thin line is the global planned path to the goal.

Several tunings of the replanning conditions have been tested to evaluate the influence of each one. The *not_moving* condition ensures that the robot is not stuck as it acts as soon the robot has reached or couldn't reach its navigation goal. However, when it is used with the *too_much_time_near_goal* condition the robot usually replans before reaching its goal, hence, it is only activated when it could not reach the goal. The *too_much_time_near_goal* condition

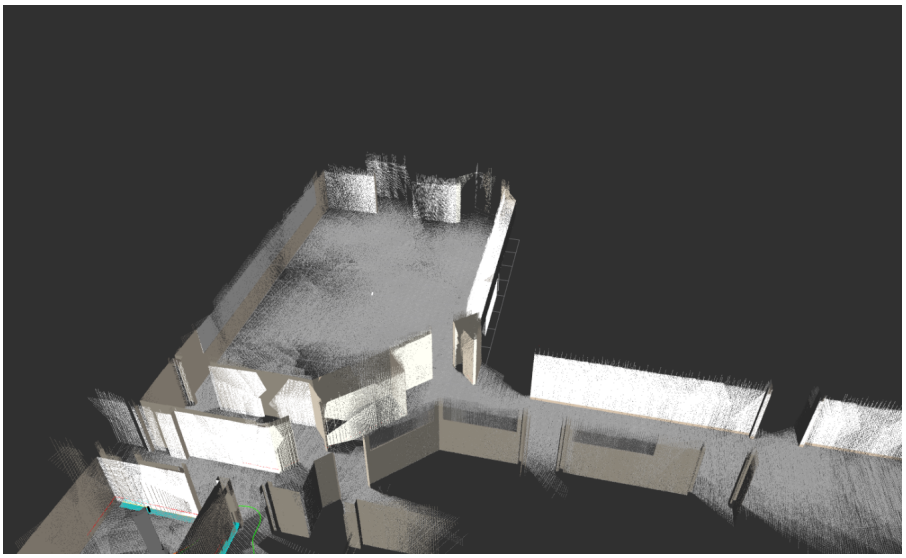


Figure 10: Sample 3D model constructed in simulation.

serves to save some time when compared with *not_moving*. Using the parameters in table 1 a new goal is planned when the current goal is no longer useful for exploration but the base controller does not confirm that the goal has been reached yet.

Parameter	Tuned value
Time threshold	0.3s
Distance threshold	0.5m
Orientation threshold	0.45rads

Table 1: Tuned parameters for the *too_much_time_near_goal* replanning condition.

The most frecuent cause of replanning is, by far, *isolated_goal*. When it is used with reasonable *depth* values (e.g. from 3 to 20, approximately), most new goals are triggered by this condition. It allows the robot to avoid stoping between goals and makes the navigation more fluent and dynamic. The value of *depth* plays a central role on the level of dynamism. Too small values for this parameter can lead to constant changes of goal without exploiting each one.

Regarding the evaluation of frontiers each evaluation function has been tested individually. Also, a number of different combinations of them have been tested. The *max_size* evaluation function usually benefits the complete exploration of the current room over the exploration of new regions, avoiding to left some unexplored areas behind. The main drawback of this function is that it does not favour the continuity of the exploration, so that there are scenarios where the robot constantly jumps from an unexplored region to a far away one if both have comparably large frontiers. Conversely, the *min_euclidian_distance* usually favors the nearby

frontiers. However, overlooking the obstacles often makes it suggest non resonable frontiers, see figure 11.

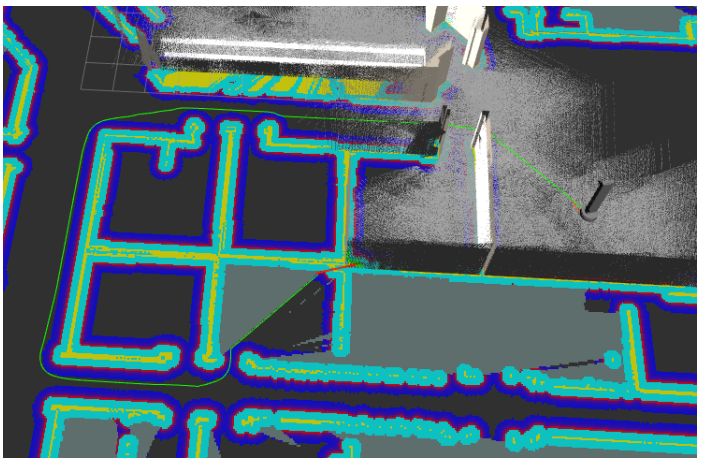


Figure 11: The *min_euclidian_distance* can favour hard to reach frontiers. The yellow and blue lines represent obstacles (walls) in the map. The green thin line is the planned path to be followed to reach the goal.

In the *min_astar_distance* the $\mathcal{A}$* measure has been tested to find short paths from the robot position to a frontier, avoiding obstacles. See figure 12 for an example of a computed $\mathcal{A}$* path. It has also been noticeable the computational burden derived from $\mathcal{A}$* computation. In scenarios where the robot has some frontiers far away from it, the evaluation of frontiers can last several seconds.

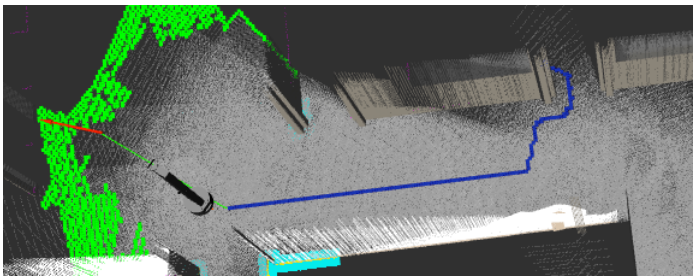


Figure 12: The $\mathcal{A}$* path from the robot position to the top right frontier is shown in blue.

In the tests with a combination of frontier evaluation functions, the *min_euclidian_distance* does not seem to improve the exploration when combined with *min_astar_distance*. The combination of *min_astar_distance* and *max_size* provides good exploration behaviour, although the $\mathcal{A}$* evaluation is computationally costly.

Regarding the goal selection, checking the visibility of the frontier from the navigation goal has substantially improved exploration and avoided situations in which the robot get stuck, as in figure 13. Goals are not equidistant of obstacles, nor are the closest point to the robot.

However, the particular point in the frontier choosen as target point has little impact as the frontier is rapidly swept, if needed, by subsequent goals.

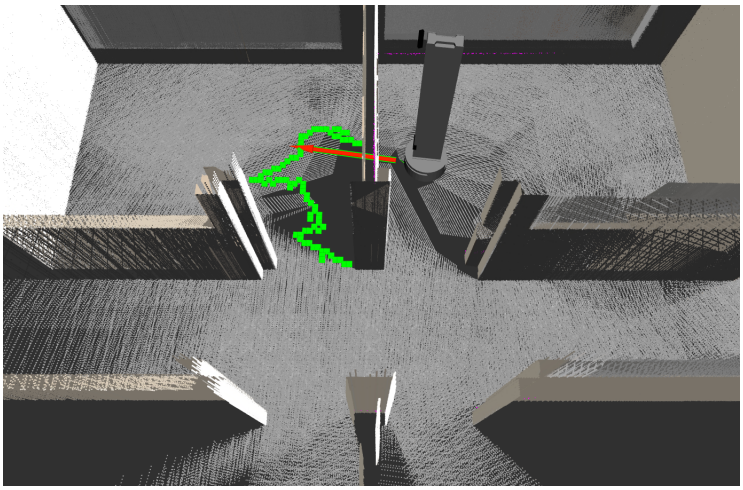


Figure 13: Example of goal chosen disregarding obstacles.

5.1.2 Real environment

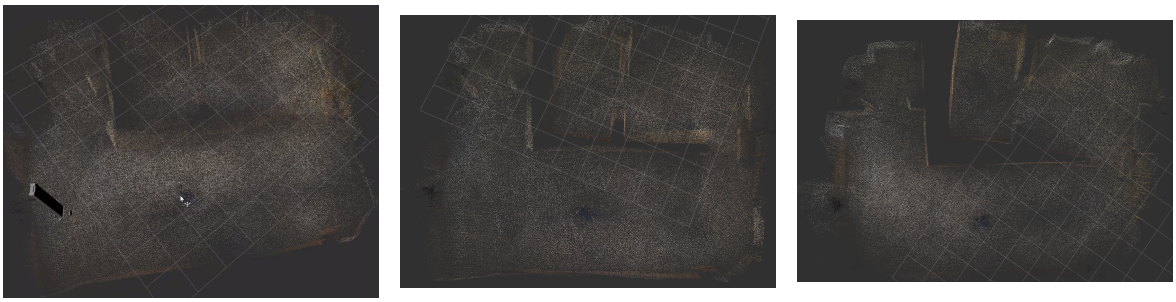
In the real robot, only a single camera is used. The testing environment consists in a circuit made of stacked boxes as shown in 14. In this case the tests performed consist on a full autonomous 3D mapping process. The tobot starts anywhere in the cricuit, knowing nothing about its environment and it autonomously builds a complete 3D map of the environment.

The table 2 shows the results of three tests with different configurations. This results are highly dependent on the actual experiment and should not be taken as a thorough evaluation of the corresponding setup. Many qualitative observations serve as well of measure of the performance of the configuration.

In the first test, the robot left some frontiers for which it had to go back. In the second test, the robot swept the area efficiently, but spent a lot of time in the decision of the last goals due to the $\mathcal{A}^*$ computations. In the third test, the robot tried to go to some unreachable frontiers which were lately discarded and achieved a complete 3D model with little traveled distance.

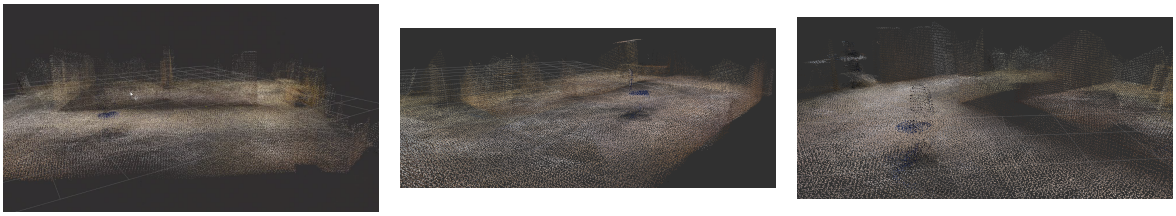
Figures 15 and 16 show captures of the whole point cloud from the top and facing the chair, respectively. Test 3 seems to represent poorly the environment when compared with the other two. The more challenging part seems to be handling the drift from the central room when the robot starts, to the top left region. The difficulties of finding loop closures and assemble sensor information is mainly a *rtabmap_ros* node concern, although the exploration has an effect on the final result. The more the robot goes back and forth and spins around, the poorer the result is.

Figure 14: Circuit made of boxes.



(a) Test 1. **(b)** Test 2. **(c)** Test 3.

Figure 15: Top view of the resulting 3D models.



(a) Test 1. **(b)** Test 2. **(c)** Test 3.

Figure 16: View of the chair in the corridor.

	Test 1	**Test 2**	**Test 3**
min_euclidian_distance	Unactive	Active	Active
weight	-	1	1
dispersion	-	1	1
min_astar_distance	Active	Active	Active
weight	1	4	4
dispersion	3	3	3
max_size	Unactive	Active	Active
weight	-	1	1
isolated_goal:depth	5	5	15
Ellapsed time	405.06	372.725	411.83
Travelled distance	84.90	71.96	58.09

Table 2: Real 3D mapping tests' parameters and results.

5.2 Hole detection

5.2.1 Simulation

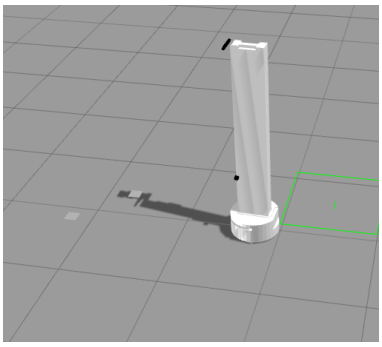
In simulation, a holed surface has been constructed and used to support the Advanrobot. The holed surface with the robot can be seen in the Gazebo GUI, see figure 17a. Two different scenarios have been tested. In the first case, the robot and the holed platform are all that exists in the Gazebo world, in the second one, a solid platform below the holed one is added so that the camera beams reach it in the holes. Both cases are shown in figures 17b and 17c, respectively. In both cases the holes are correctly detected and placed (see the isolated squares upon the holed surface). Also, in both cases the local costmap includes the holes so that they are avoided when plannning (see the yellow and cian regions). In the latter case, the actual sensor readings can be seen as squares below the ground surface.

5.2.2 Real environment

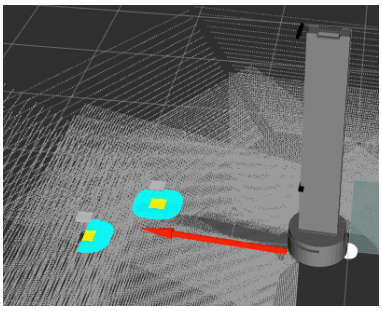
Two different scenarios have been used to test the performance of the hole detection tool.

Advanrobot facing a window

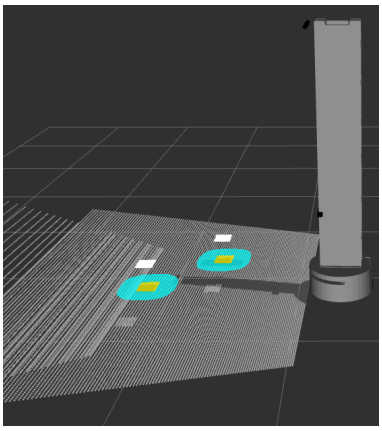
To test the NaN handling, the Advanrobot has been located near an office window, facing it. This is a tipical cliff-like scenario which can be used to test the hole detection behaviour under



(a) Platform holes in the Gazebo GUI.



(b) Detected holes where there is nothing below.



(c) Detected holes with another surface below.

Figure 17: Hole detection simulation with the robot mounted on a holed platform.

similar scenarios, as in the case of a descending stairs. Under this conditions, the mounted RGBD camera is able to capture the wall below the glass but most of the points correspond to the glass. Those points can either return a NaN or the lecture corresponding to a reflected beam (a mirror effect). Figure 18 reflects this situation. Aside from the direct lecture from the wall and the ground, some points in the cloud are located as if there was a room symetric to the actual one on the other side of the wall. There are reflected points due to the ground and some comming from the robot itself.

It is noticeable that beyond the wall are not densely representing the room but there are missing regions. All the missing points, which correspond to NaNs should be recognised by the hole detector facility.

In figure 20 the result of processing this point cloud with the hole detection utility is shown. Each white point corresponds to a missing point in the original point cloud that could have been reconstructed with the information from a recorded view.

As can be seen, the main gaps in the output cloud correspond to regions where there have been some reflection, due to the proximity of objects in the other side (robot body) or due to the reflectivity of the surface (the metallic bottom part of the window).

The same test has been made with different light conditions, leading to significanly different results. Figure 20 shows the results of a test with dimmer light conditions.

Visible holes

To test the performance of the hole detector, the robot hass been moved on a pallet, so that the flat floor acts as a hole, see image 21a. The expected behaviour is that the lower (floor) points

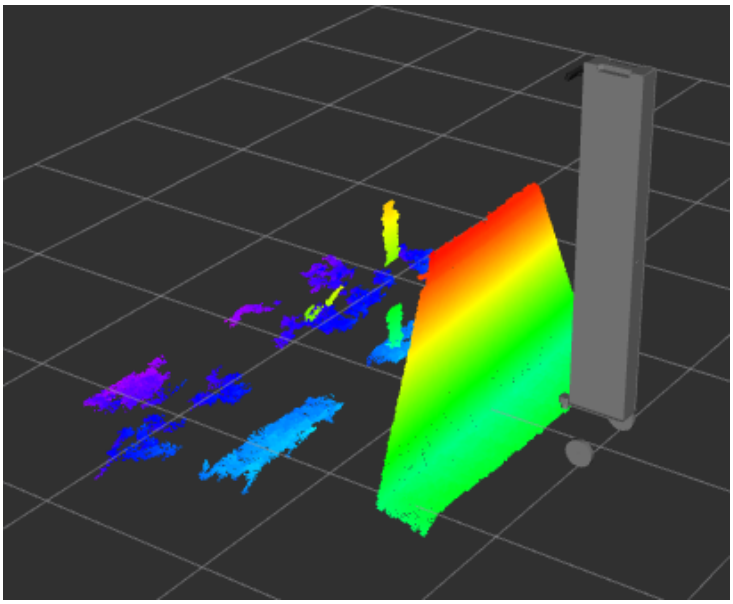


Figure 18: Point cloud with the robot facing a window. Midday test.

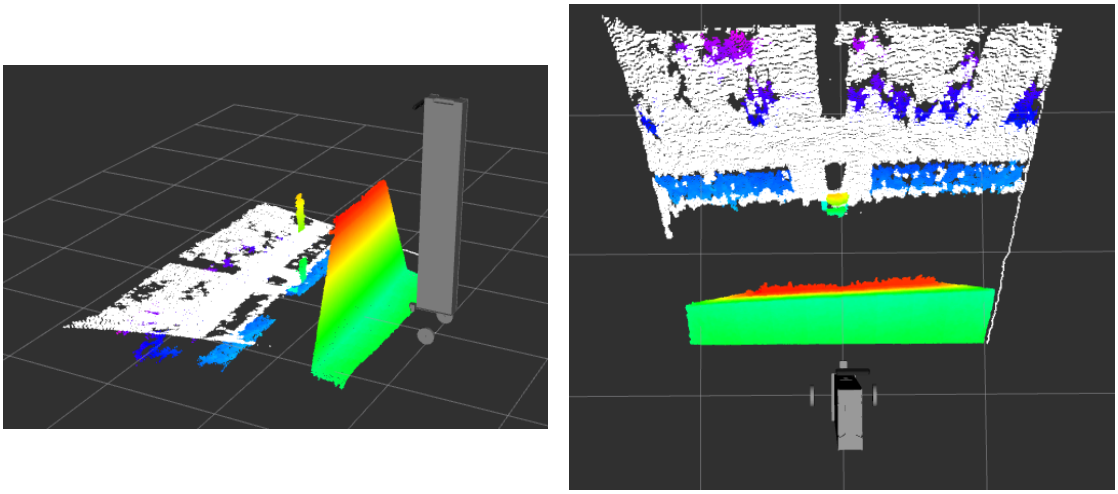


Figure 19: Hole detection results for the Advanrobot facing the window. Midday test.

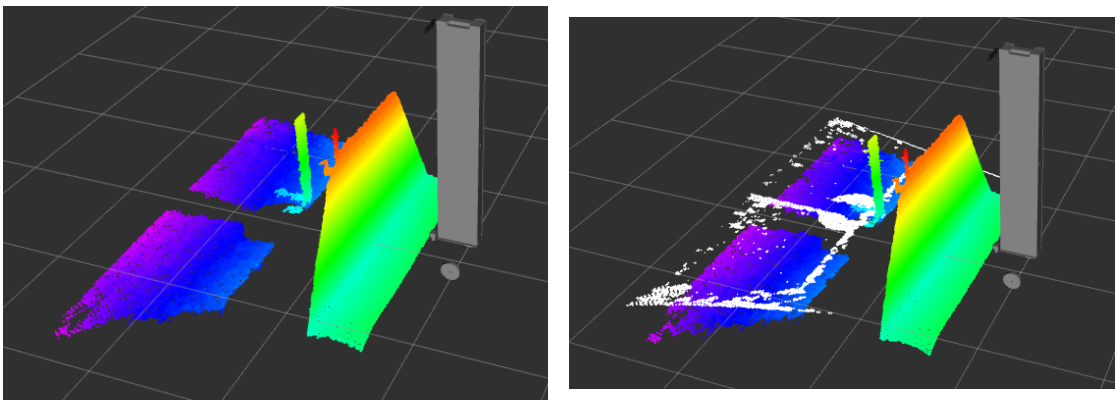
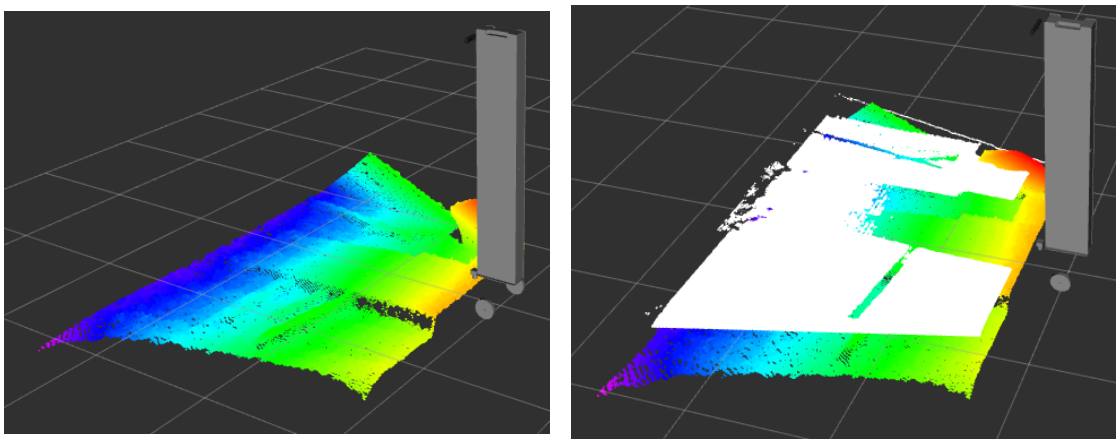


Figure 20: Hole detection results for the Advanrobot facing the window. Morning test.

are recognized and obstacles corresponding to each hole point are then properly located. As it can be seen in figure 21b, the holes are correctly detected. The beams that hit the upper surface of the closer wooden bars are not detected as hole points, because of their height.



(a) Original point cloud.

(b) Detected holes.

Figure 21: Test with the Advanrobot on a pallet.

5.3 Point cloud calibration

As shown in figure 4, the data comming from the RGBD camera when facing the ground, diverges from a plain. Figure 22 shows both, original and corrected point clouds. The lower regions in the original point cloud appear now in white, as in those points the corrected point cloud is above the original one.

Also, the corrected point cloud looks more flat than the original one. To see this, compare figures 23 and 4.

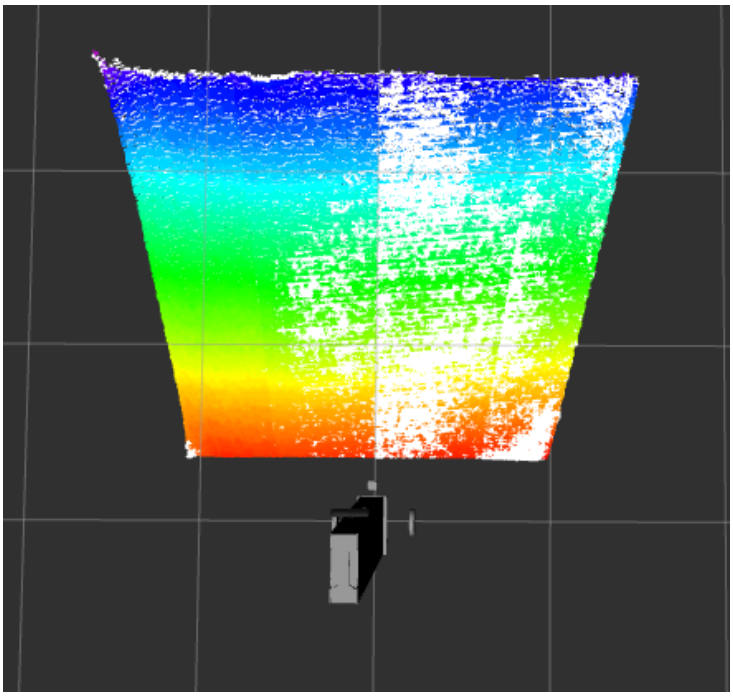


Figure 22: The original point cloud is rainbow-colored. The corrected one is white. They overlap so that there are regions where one is on top and other regions where it is below.

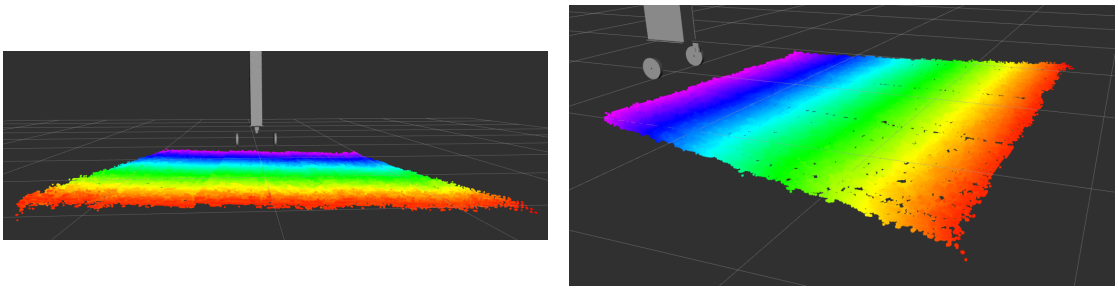


Figure 23: Corrected pointcloud.

6 Sustainability and social impact

The inventory tasks are usually tedious and require a number of people to proceed manually. For large stores, human teams have to spend many hours with manual inventory. That time tend to be at night, when stores are closed to customers. By contributing to the development of the Advanrobot, this projects helps to put forward the automation of the inventory process, thus reducing costs and time.

This projects also contributes to 3D model building with a robotic platform, which has already been proved to have a great impact in domestic robots as is the case of the Roomba vacuum cleaner. One could imagine a future scenario where domestic robots capable of generating 3D models of the environment are mainstream.

6.1 Socioeconomic impact

The cost reduction involved in automating the inventory process in large stores is straightforward. The massive adoption of these systems needs to rely on a mature and complete project. This work pushes the Advanrobot one step forward towards the maturity of the project.

Such automation would yield reductions of inventory staff in large retailers as the same work would be done with less people handling mainly supervision tasks. Of course, the introduction of automated inventory would open new possibilities for technical staff developing the autonomous systems.

Regarding the ability of producing reliable virtual stores based on 3D models of the actual one is demanded by retailers. Providing a tool to develop virtual stores would not only help retailers in providing a better customer experience, but also allow new ways of customer-retailer communication.

Virtual stores based on 3D models would change the way we buy products allowing faster, agile and reliable buying experience opening new possibilities from the customer viewpoint.

6.2 Environmental impact

The deployment of Advanrobot comes with the environmental concerns associated to its manufacturing and transportation processes. While engaging the development of the product one contributes to the potential production of batteries, motors and other pieces of hardware that compose the final product. Also the robot is electrically powered increasing its environmental impact.

However, the automation of the inventory in retail stores diminishes the use of handheld

readers, and reduces the needed staff with the corresponding energy consumption at work.

7 Publications

The work of this thesis is part of the Springer's *Robot Operating System (ROS) Volume II* book, chapter *Development of an RFID-based inventory robot (AdvanRobot)* which is accepted and is to be published. The code shared according to the chapter publication policy corresponds to the work described in this document and can be found in the public repository:
https://github.com/UbiCALab/cam_exploration

8 Conclusions

Some conclusions from the work in this project are listed below:

- Data comming from standard RGBD cameras is much more noisy than the one from standard LIDAR sensors.
- The treatment of point cloud messages can lead to a high computational burden even if performing simple point-wise operations.
- The introduction of a second camera allows a much wider scanning range, which fits the needs of a wheeled robot with fixed camera(s).
- A frontier based exploration approach has been proven to properly work with point cloud data.
- 3D models can be effectively build with unexpensive hardware.

9 Future work proposals

Below there is a list of future work proposals that have not been developed in this work because of the time limitations.

- Adding more configuration possibilities (replanning conditions, frontier evaluation functions and goal decision functions). Some proposed replanning conditions are:
  - Replanning when a new obstacle appears between the navigation goal and the frontier target point.
  - Replanning when the trajectory to the goal has to be replanned as it implies that changes have been made that affect the relation between the robot and the navigation goal (e.g. a new obstacle between both points).

  Some proposed frontier evaluation functions:
  - Prioritize frontiers that are ahead from the robot over those that are in the back or in the sides of the robot.
  - A distance-based evaluation functions that takes as distance the plan to the frontier target point. This would be the most reliable distance measure but it would imply a high computational burden.

  Some proposed goal evaluation functions:
  - Choosing the frontier target point which is near to the robot position according to some distance measure.
  - Variations of the goal evaluation functions that take into account the proximity between the frontier target point candidates and the nearby obstacles. Frontier target points with no obstacles in their neighbourhood should be better evaluated since they make the problem of finding a proper navigation goal, simpler.

- Building a benchmarking system to properly evaluating exploration performance measures such as:
  - The presence of outliers in the resulting 3D model.
  - The time spent building the 3D model.
  - The robot travelled distance.

  Such system could compute the effect of each configuration parameter (or combination of them) in the previous measures.

- Develop machine learning strategies that learn which action to take (regarding replanning, frontier evaluation and goal selection) according to the already build functions taken as inputs and the performance measures previously stated.
- Adapt the RGBD camera calibration tool to fit a plane according to many point cloud samples instead of relying in a single one.
- Extend the RGBD camera calibration tool to correct the robot description (URDF file) by matching the fitted plane with the theoretical position of the ground.
- Change the hole detection tool in order to implement it as a costmap plugin using the ROS pluginlib. That would allow greater flexibility and make the tool easier to use.
- Add a blacklist of frontiers, so that when the robot gets stuck in a certain point it avoids sending navigation goals near such point.

10 Aknowledgements

I would like to thank both Keonn Technologies and Universitat Pompeu Fabra for the opportunity they have given me to work with their project Advanrobot. Especial thanks to Rafael Pous, Victor Casamayor and Marc Morenza for their patience and support.

11 References

[1] DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.

[2] DORNHEGE, C., AND KLEINER, A. A frontier-void-based approach for autonomous exploration in 3d. *Advanced Robotics 27*, 6 (2013), 459–468.

[3] EITAN MARDER-EPPSTEIN. ROS navigation stack.

[4] FISCHLER, M., AND BOLLES, R. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM 24*, 6 (1981), 381–395.

[5] GERKEY, B. P., AND KONOLIGE, K. Planning and control in unstructured terrain. In *In Workshop on Path Planning on Costmaps, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA* (2008).

[6] LABBE, M., AND MICHAUD, F. Appearance-based loop closure detection for online large-scale and long-term operation. *Robotics, IEEE Transactions on 29*, 3 (2013), 734–745.

[7] LABBÉ, M., AND MICHAUD, F. Memory management for real-time appearance-based loop closure detection. In *IROS* (2011), IEEE, pp. 1271–1276.

[8] LABBÉ, M., AND MICHAUD, F. Online global loop closure detection for large-scale multi-session graph-based slam. In *IROS* (2014), IEEE, pp. 2661–2666.

[9] LEE, D.-Y., LU, Y.-F., KANG, T.-K., CHOI, I.-H., AND LIM, M.-T. 3d vision based local obstacle avoidance method for humanoid robot. In *Control, Automation and Systems (ICCAS), 2012 12th International Conference on* (2012), IEEE, pp. 473–475.

[10] MAIER, D., HORNUNG, A., AND BENNEWITZ, M. Real-time navigation in 3d environments based on depth camera data. In *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on* (2012), IEEE, pp. 692–697.

[11] MENG, L., DE SILVA, C. W., AND ZHANG, J. 3d visual slam for an assistive robot in indoor environments using rgb-d cameras. In *Computer Science & Education (ICCSE), 2014 9th International Conference on* (2014), IEEE, pp. 32–37.

[12] RUSU, R. B., AND COUSINS, S. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)* (Shanghai, China, May 9-13 2011).

[13] SENARATHNE, P. G. C. N., AND WANG, D. Frontier based exploration with task cancellation. In *SSRR* (2014), IEEE, pp. 1–6.

[14] SUZUKI, K., HORIBA, I., AND SUGIE, N. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding 89*, 1 (2003), 1–23.

[15] YAMAUCHI, B. A frontier-based approach for autonomous exploration. In *CIRA* (1997), IEEE Computer Society, pp. 146–151.