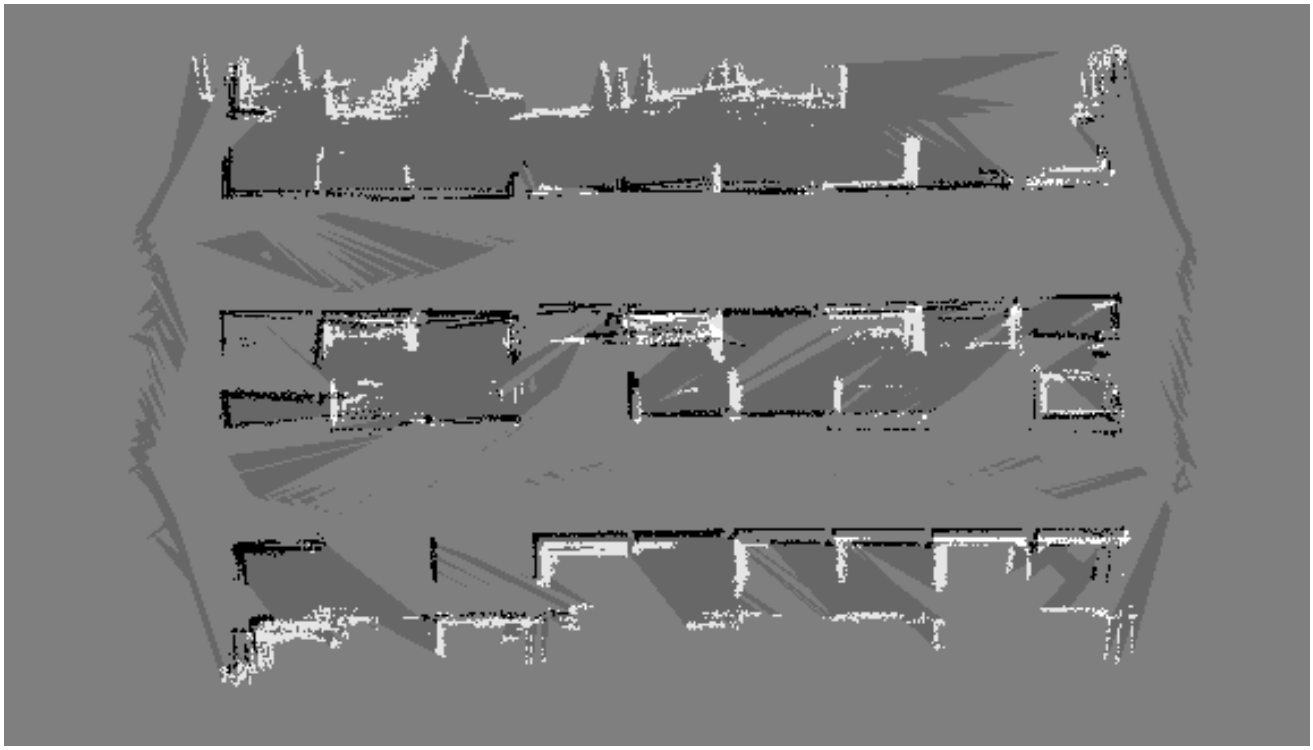
CENTRALE NANTES

Map drift during SLAM

Alexis Dupuis, Corentin Chauvin-Hameau

2018



Contents

1 Presentation of the problematic

The problem studied here is the control of an autonomous mobile robot present on a containers-stocking area. The robot localises itself thanks to SLAM methods and will have to plan its trajectories thanks to the map it builds.

Since the environment is constantly changing (containers appearing and disappearing), one may think that the map could derive. The goal of this project is to implement a simulation experiment in order to show (or not) the drift of the map. The robot used is not as important as the methods of localisation and mapping, this is why we will use a Turtlebot instead of the concerned robot. The Turtlebot robot is a (2, 0) mobile robot which is very easy to control, and which already has a simulation implemented in Gazebo.

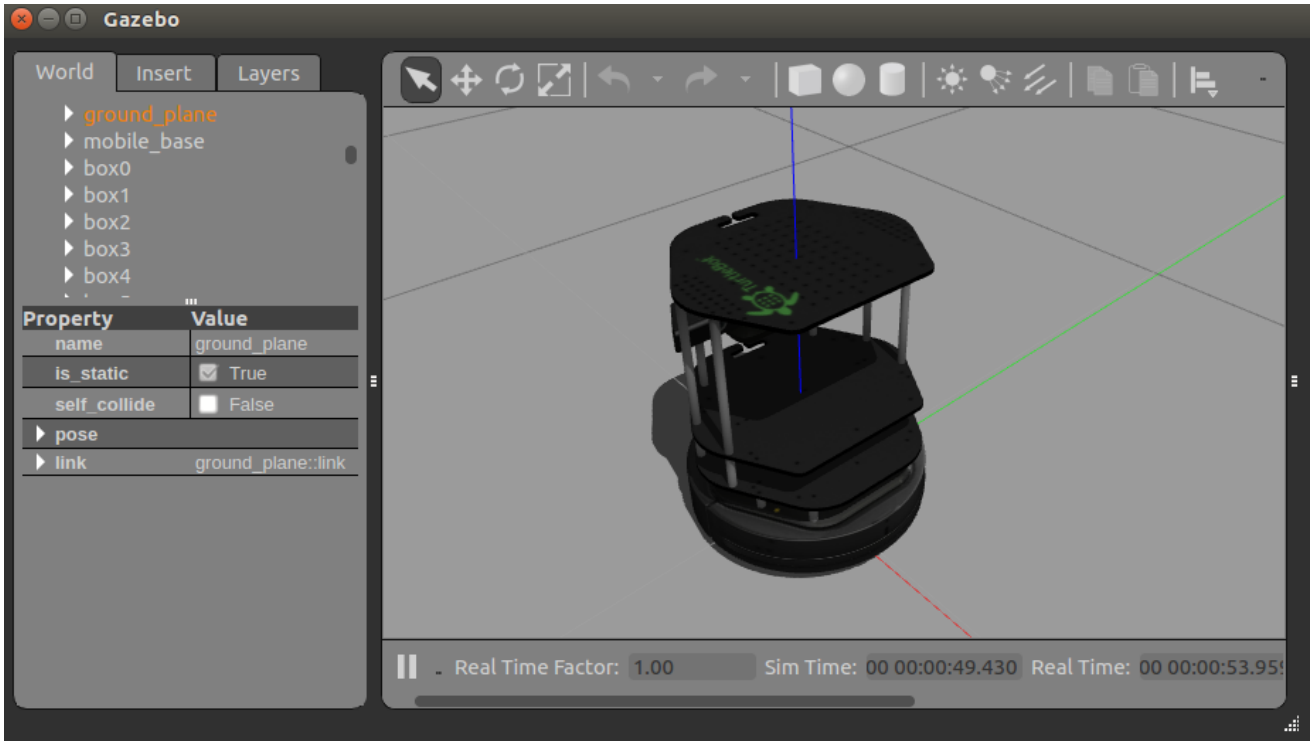


Figure 1: Turtlebot simulation on Gazebo

We are going to present the SLAM methods, and especially the one used by the robot, in order to introduce in more details the issue at stake in this project.

1.1 Short introduction on SLAM

The SLAM (for Simultaneous Localisation and Mapping) is a very active research field since the 1990's especially in robotics. It consists of the combination of two main goals:

- **Localisation**: the ability to get the pose of robot on a map

- **Mapping**: the ability to construct and modify a map of a given environment

The difficulty of SLAM comes from the simultaneous realisation of both these tasks.

The robot needs to receive information from proprioceptive and exteroceptive sensors. The proprioceptive information in our Turtlebot simulation is the computed odometry, while the exteroceptive information is ensured by a simulated laser scan.

The mapping operation builds a metric map thanks to the sensors. These are complete laser scans, so that the map is only constructed upon simultaneous scans of the environment without the need to detect specified objects (or "amers") such as beacons. The detected obstacles are expressed in the robot's frame with the exteroceptive sensors information and reported on the map's frame with respect to the estimated robot's pose. To sum up, the trajectory of the robot determines the incremental building of the map, which in turns gives full information on the trajectory of the robot.

The algorithms used take into consideration the errors in position of these elements, usually represented in a covariance matrix (like for a Kalman Filter). The space is represented by an occupation matrix (basically a fine grid of the environment) which represents the probabilty of presence of obstacles. This is the case with `gmapping` (the SLAM method we will use during this project) in which an unknown space of the environment is represented in dark ray, an obstacle in black and a free space in white.

There are 3 main categories of SLAM algorithms:

- Those based on the **"correlation of the measurements"**: we suppose the perceptions at successive times to be close enough so that they contain elements in common. These elements are used to infer the position of the environment detected at $t + 1$ from the one detected at t. Proprioceptive information is not mandatory in this case.

- **"Filters"** which use proprioceptive information to estimate the robots pose and exteroceptive information (even partial) to correct it, such as the Kalman filter in SLAM.

- **"Optimisation methods"** which use the full stack of information accumulated to compute the map that best reduces constraints based on correlation of the measurements and proprioceptive information. Those methods are generally more complex.

The robot should also be able to use the map to plan a trajectory. This is ensured by a global planner which generates paths from a start point and a goal, only using the map. It is coupled with a local planner whose purpose is to prevent collisions with close obstacles and modifying the planned trajectory locally if the information of perception doesn't match the expected environment (based on the map).

Both planners have their costmap, build upon information on the map or given by the sensors. The local planner is a rolling window, which means it is always centered on the robot. The global planner can be either a rolling window or a static map (whose centre is a fixed point). In some cases, the global costmap can vary in size as the robot discovers the map.

Finally, we must draw attention on a particularly hard issue in SLAM called the **"closed loop detection"**. If a robot comes back to a place and can't perfectly match the two sets of

information he has from the two moment he was there (which isn't done naturally because of the errors on proprioception and exteroception), the map build will be very incorrect when the mapper will try to close the loop. For this reason, a special attention is given to the ability of a SLAM algorithm to execute a closed loop detection.

1.2 Rao-Blackwellised Particles Filter (RBPF)

The Rao-Blackwellized Particle Filter is a filter method for SLAM, and is in particular the method used on the robot. It uses odometry as a pose estimation. To correctly explain how it works, we have to explain how a filter method works in SLAM.

As explained earlier, a filter method uses a combination of proprioception and exteroceptive sensors. The global methodology is to alternate:

- A prediction phase in which a function f describes the evolution of the state thanks to proprioception and a function h predicts the perceptions from the state.

$$X_t^* = f(X_t, u_t) \text{ and } Y_t^* = h(X_t^*)$$

- A correction phase that computes the estimated state from the predicted state and perceptions, by also considering the covariances (uncertainties). In the equation below, the "gain" K is the one computed from the uncertainties on proprioception and perception. The covariance matrices are then also corrected.

$$X_{t+1} = X_t^* + K(Y_t - Y_t^*) \text{ and (for the Kalman filter) } P_{t+1} = P_t^* - KHP_t^*$$

Where P is a covariance matrix and H is the Jacobian of h.

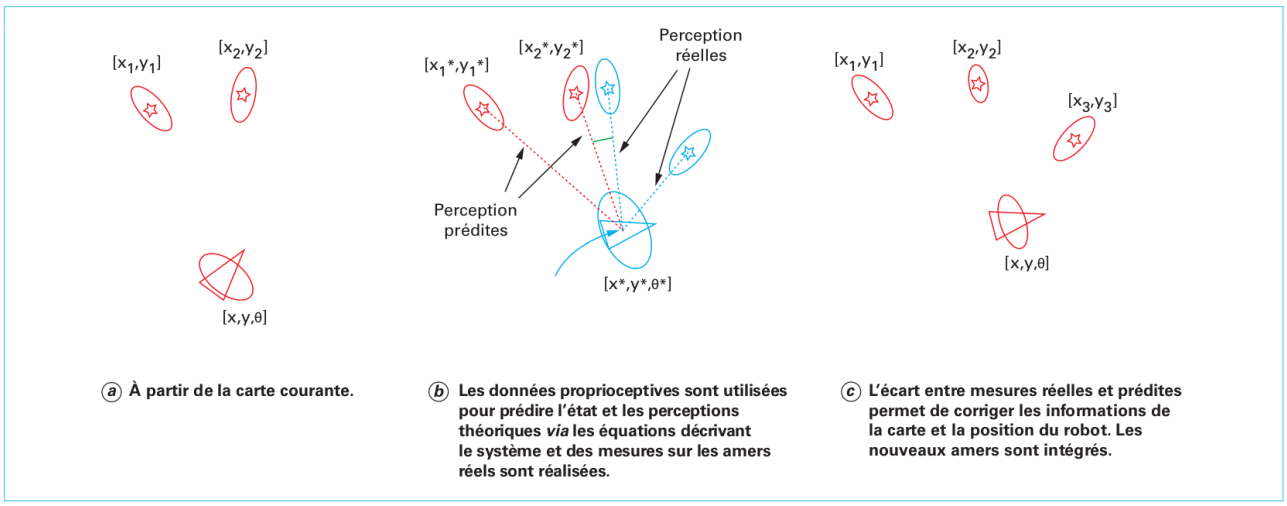


Figure 2: Illustration of a SLAM filter [1]

Amongst the filter-based methods, we distinguish the particles filters. Instead of using a gaussian approximation of the state, we use a set of **"particles"** that embodies the state-space. In SLAM, the state-space is the set of all possible trajectories. The probability for a particle to hold the best version of the state is given by $f = w * d$, where w is the weight given to that particle by a weight function, and d is the density of probability. The particles allow to represent any distribution in a compact manner, instead of only normal distributions.

Each of these particles (ie predicted trajectories of probability f) are used to build a map with the perceptions (which is done trivially since every particle is supposed to holds the perfect trajectory at that moment). The weight of each particle is then re-evaluated according to its plausibility (the latter being especially easy to determine during a closed-loop detection). A resampling of the particles is done regularly to drop those of low weight and duplicate those of heavy weight.

The **Rao-Blackwellisation** of a particle filter consist in separating the very non-linear components of the problem (to be treated by a particle filter) from the less complex ones (to be treated by a classical Kalman filter for instance). This allows to reduce the computation complexity and the memory required. Thus, RBPF are especially useful for big maps.

In particular, `gmapping` uses a RBPF with the particularity that a resampling of the particles is only done if computationally needed to prevent a shortage of particles.

1.3 The issue of SLAM on evolutive maps and large environments

SLAM algorithms are particularly performant on static maps, but issues arise when the robot evolves in a dynamic environment. Doors that may be open or closed, moving people or objects are some of the problems that many searchers have been proposing solutions to.

The most basic approaches use a **"forgetting factor"** that allows the progressive erasure of obstacles if they are not detected where they were previously present. Many other methods, generally of the form of an amelioration of the most famous SLAM algorithms, have been proposed and tested with encouraging results [2]. Those methods mainly consider the problem of an indoor robot, moving in an office for example. This is a severe restriction of the more global issue of SLAM on evolutive maps, where many hypothesis (unverified in our case) restrained us from using this knowledge in this project.

Aside from this issue is the one of SLAM on large environments that is usually very memory and CPU demanding. Improvements based on the Rao-Blackwellized particle filter have also been proposed. Yet, we didn't need to be much concerned by that in so far as the size of the maps used for our experiment was limited (essentially for time reasons, as moving the robot in large environments takes lots of time).

1.4 Context and stakes of the project

This project was started in the context of studies about robots dedicated to carry containers in harbours. These wheeled vehicles must navigate in an environment that changes from day to

day. Indeed, the concerned stocking area is full of containers at the begining of a week, and is being emptied during the following days.

A map of the environment is first created thanks to a laser scanner and a very fine localisation system (a RTK GPS). The resulting map is then shared to the robots working in the harbour. These robots can't use the same localisation system since GPS is not reliable between high rows of containers (the signal can easily be lost, or worst, can be reflected by the metallic surfaces, which distorts the measurements).

Therefore, since the robot finds its way only thanks to the map and its distance to obstacles, one may think the map is going to shift with respect to an absolute frame. In our case, that problem is critical since the waypoints are given in the absolute frame but followed by the vehicle on the map.

During this project, we focused on creating a simulation environment to test this problematic virtually in many circumstances. We tried to provide a realistic experiment as well in terms of access to useful tuning parameters, as in terms of closeness to the real situation of the robot; and that would be easily visualizable through ROS Gazebo and Rviz.

We worked a lot on the challenge that is the conduct of an experiment that long, trying to push Gazebo (and our CPUs) to their limits; and automatizing every task such as the giving of waypoints or the saving of maps at different times of the experiment. We also made the environment himself tuneable, with a fairly quick way to create a new environment from an image.

This project required a deep understanding of ROS, Gazebo and Rviz, but also a great understanding of `move_base` and `gmapping` and other ROS tools related to the Turtlebot.

We were able to obtain some experimental results in the end, but since we lacked time to make a sufficient number of tests, our results analysis will be made very cautiously.

2 Presentation of our ROS package "evolutive_map"

2.1 Organisation of the package

The package is composed of numerous nodes working with each other. Most of these nodes were already implemented in other ROS packages. They concern the simulation of the Turtlebot, as well as the SLAM and the navigation. Our role was to make it work together, and add our own nodes among them.
Let's begin by a short description of the already existing nodes.

2.1.1 gazebo

Handles the physical simulation of the Turtlebot and its environment. The Turtlebot is fully modeled in 3D, and the behaviour of its sensors and motors are generated by Gazebo. This is the only node that has to be removed to use the package on a real Turtlebot.

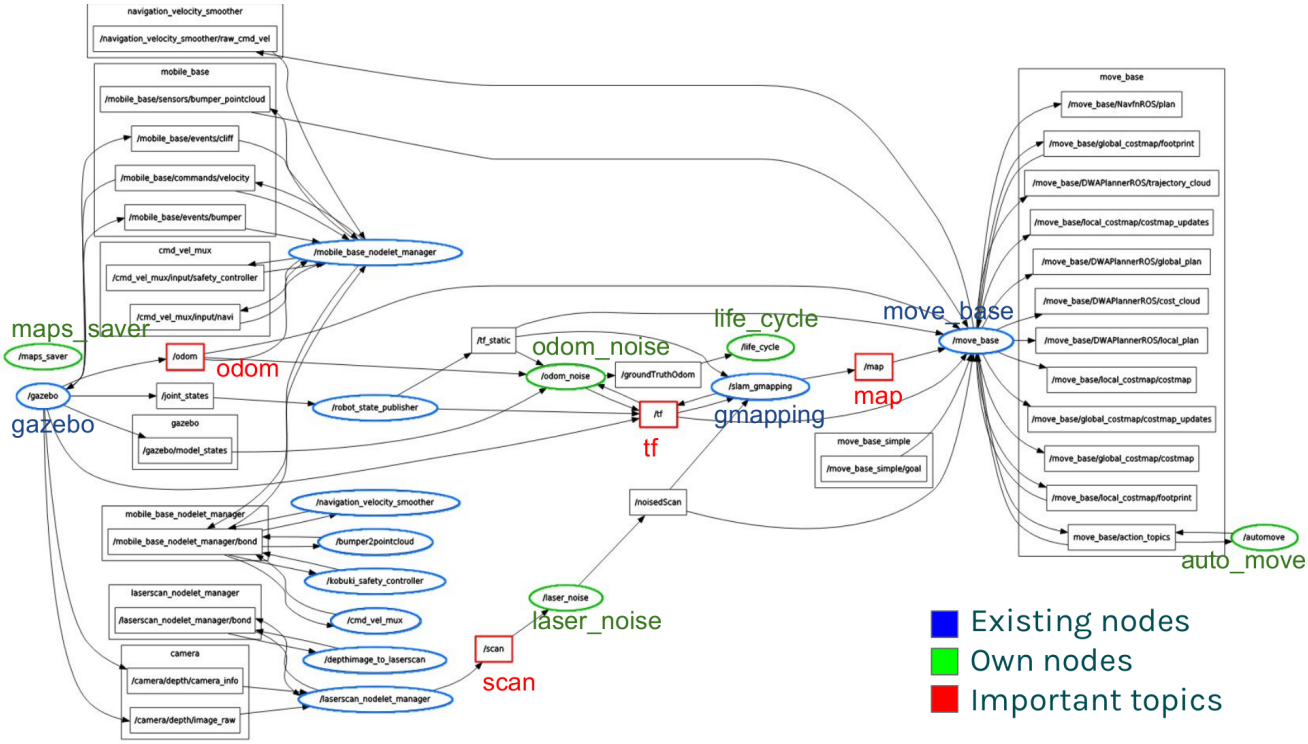


Figure 3: Full rqt_graph during an experiment

2.1.2 robot_state_publisher [3]

Publishes the tf transforms [4] of the Turtlebot, which is required by other nodes such as `move_base` and `gmapping` (see Figure 4). Even if this robot is rigid, it has many frames attached to it. The one we use at most is `base_footprint` which is at the root.

2.1.3 mobile_base_nodelet_manager

Handles everything about the mobile base of the Turtlebot. It detects bumper events and cliff events (whether the robot is still on its wheels), and gets velocity commands.

2.1.4 kobuki_safety_controller [5]

This node makes the robot act when its bumper or cliff sensors are triggered. For instance, if it bumps against a wall, the Turtlebot will stop and go a bit backward.

2.1.5 bumper2pointcloud

Publishes the data of the bumper sensor as 2D points.

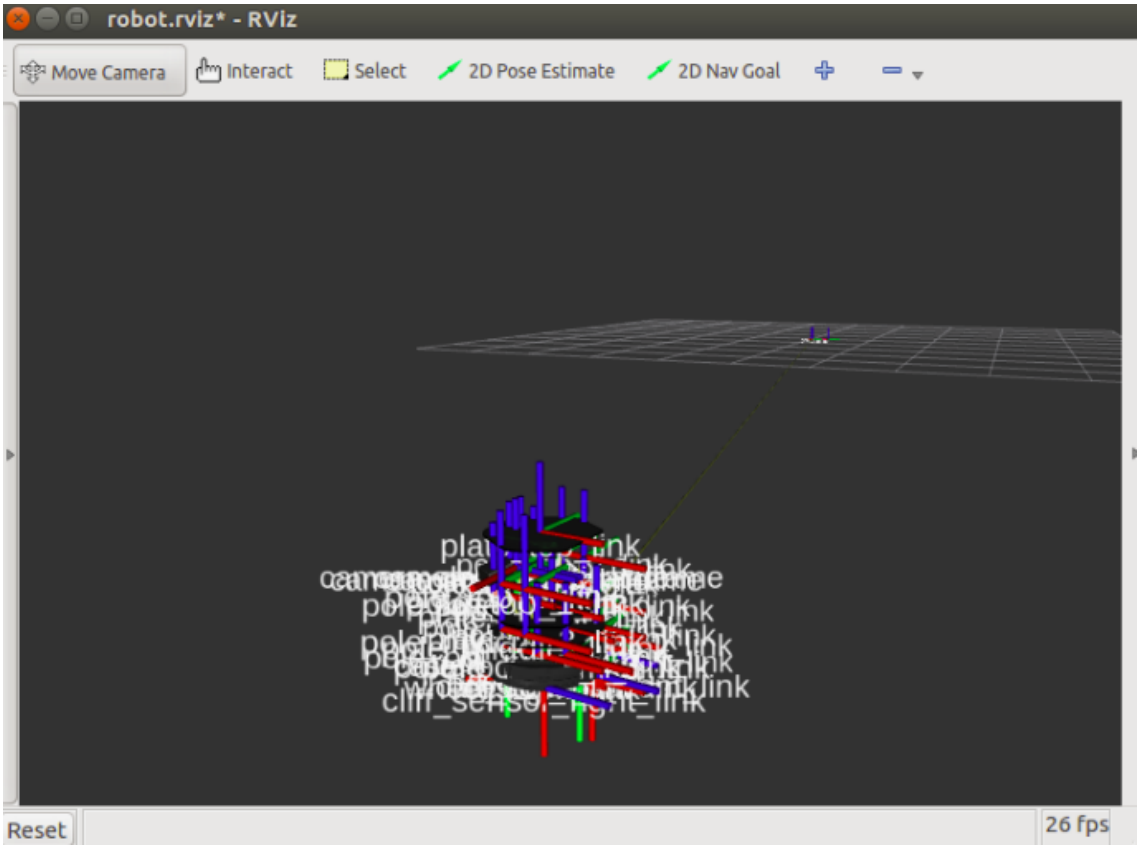


Figure 4: tf frames published by `robot_state_publisher`

2.1.6 cmd_vel_mux [6]

Acts as a multiplexer for different velocity command sources, considering priorities between the sources.

2.1.7 depthimage_to_laserscan [7] and laserscan_nodelet_manager

The distance sensor by default on the Turtlebot is a Kinect. These nodes retrieve the depth image taken by the Kinect and convert it to a 2D laser scan message by only keeping an horizontal line in the image (see Figure 5).

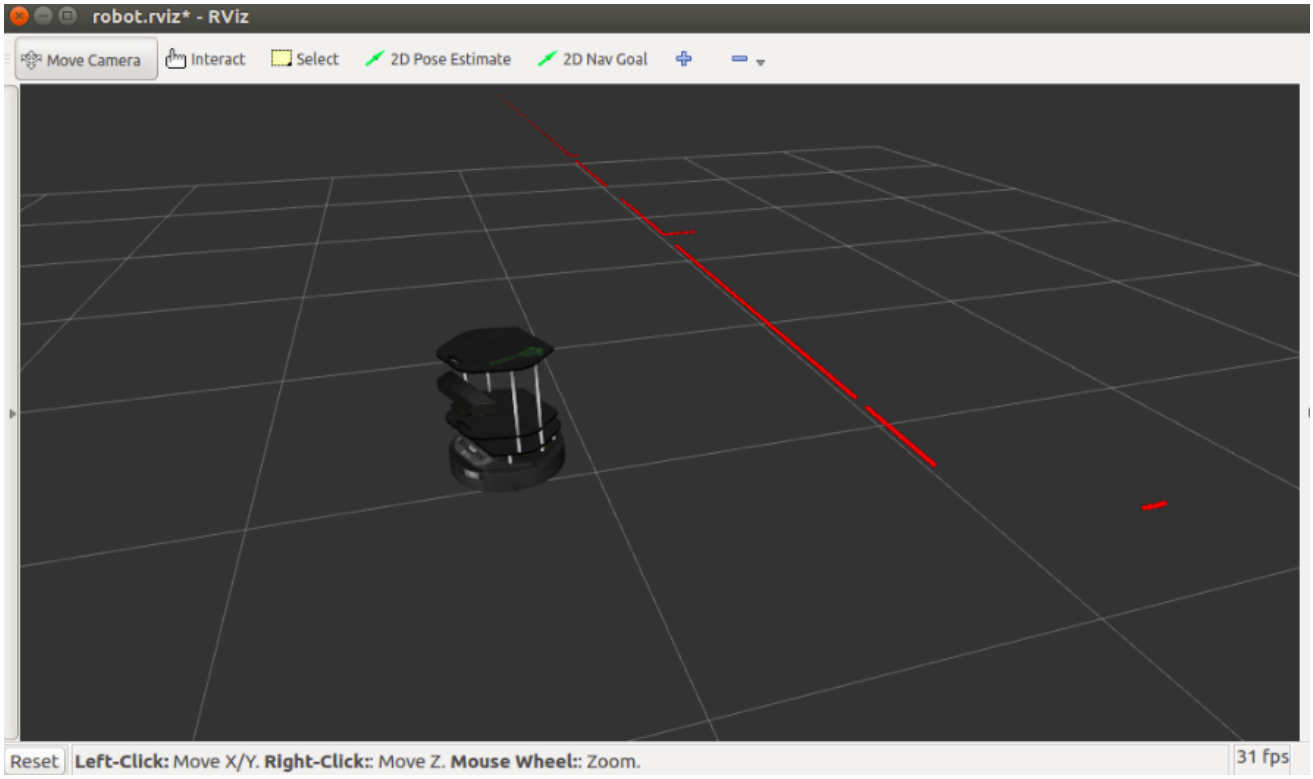


Figure 5: Visualisation of the laser scan on Rviz

2.1.8 navigation_velocity_smoother

Filters the velocity commands so that it can effectively be performed by the robot, taking into account maximum velocities and accelerations.

2.1.9 slam_gmapping [8]

This big node performs the SLAM according to the Rao-Blackwell algorithm. To work, it requires odometry and laser scan data, and tf transforms locating the laser and the coders to the mobile base. It will at the same time create a map (published in `/map`, see Figure 6) and publish the localisation of the robot (*e.g.* the pose of the `/map` frame).

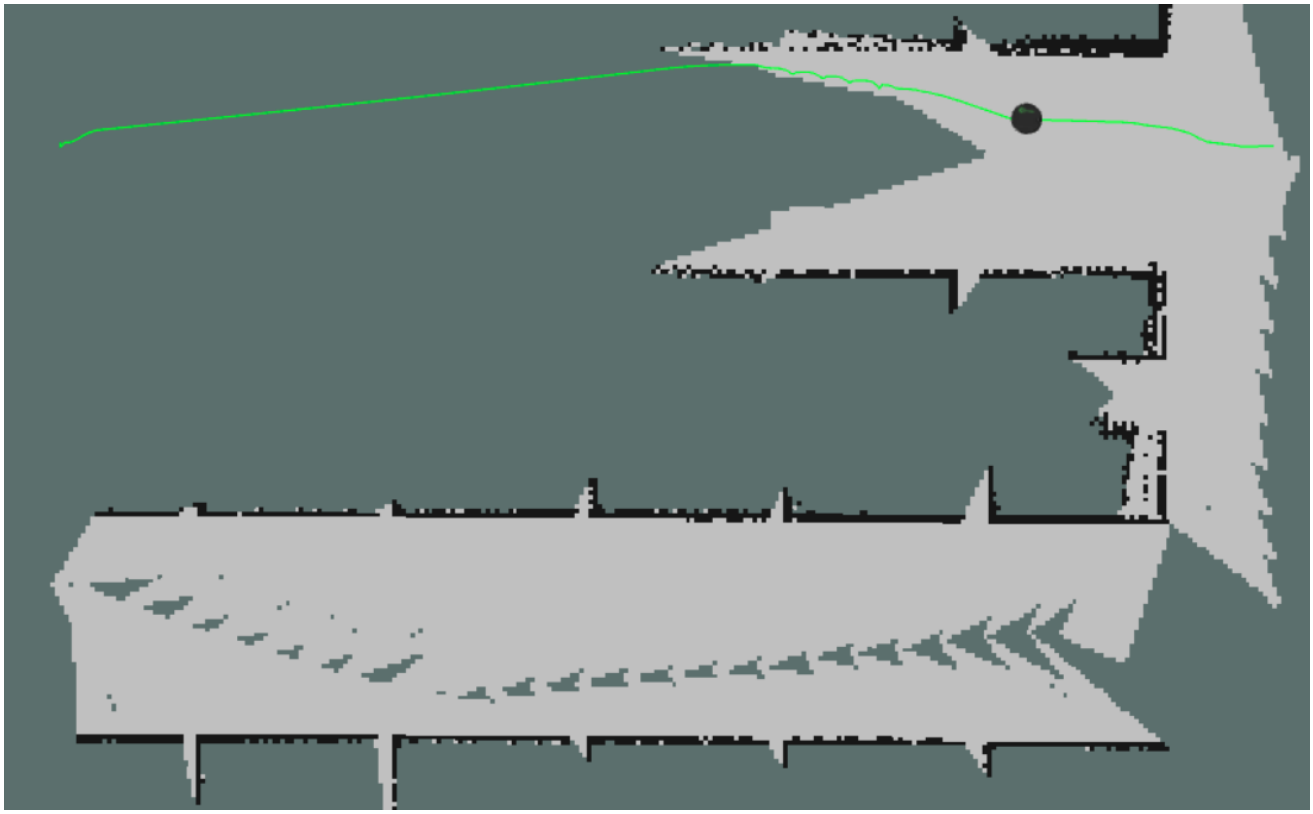


Figure 6: Map construction with `gmapping`

2.1.10 move_base [9]

This other big node handles everything about autonomous navigation. Given a pose to reach, it computes a trajectory (global planning) and sends velocity commands to the robot, while avoiding obstacles in real time (local planning) (see Figure 7). The global and local planner used during this project are respectively the `navfn` and `dwa_local_planner` plugins for `move_base`.

This node is very modular since it is designed to work on every type of mobile robots. Therefore, it can use data provided by many sensors. Though, it requires a lot of different things to be implemented to work, this is widely detailed in the ROS documentation.

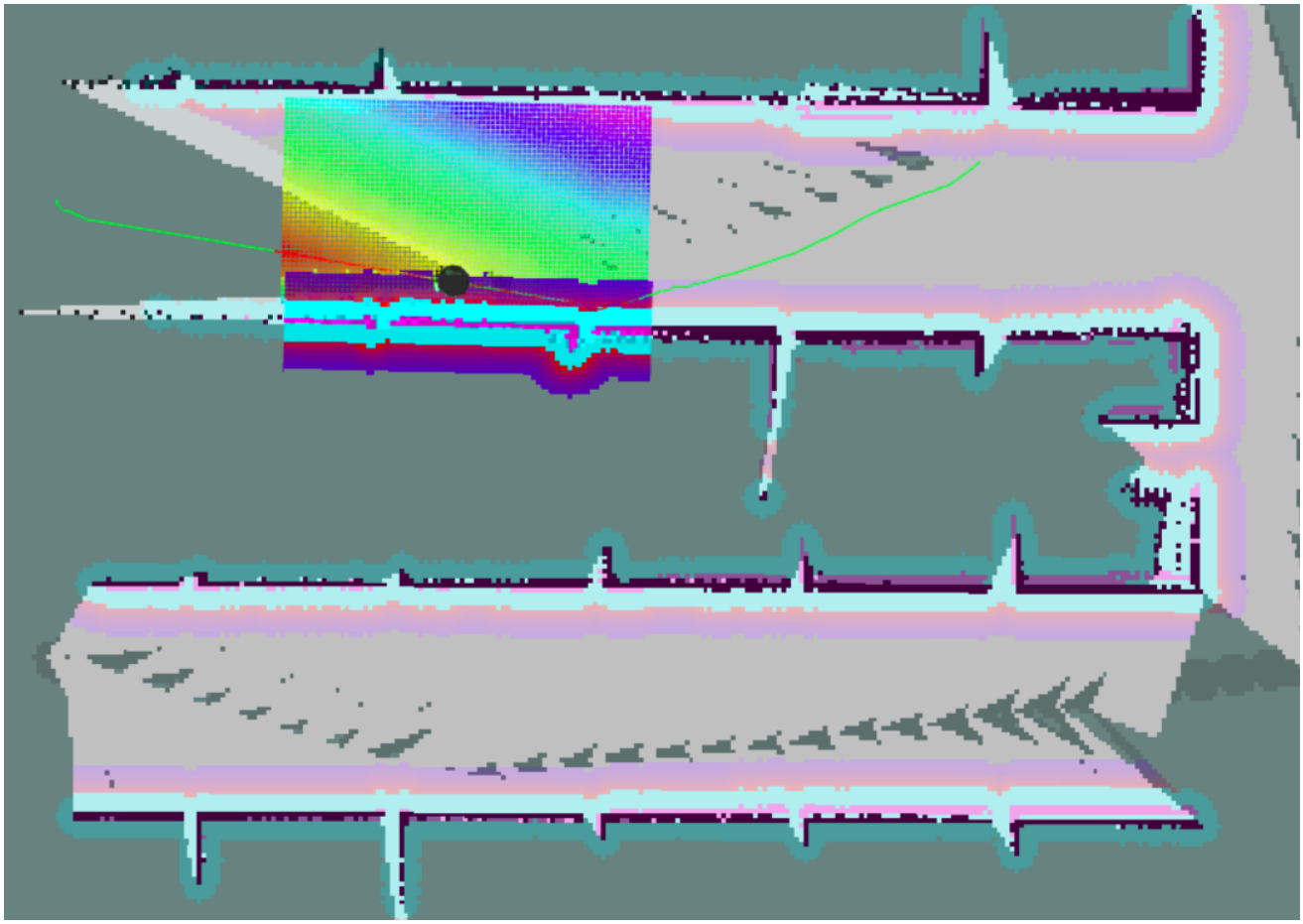


Figure 7: Navigation with `move_base`. We can see the global path and the different costmaps.

The next nodes are the one we developped for the sake of the project.

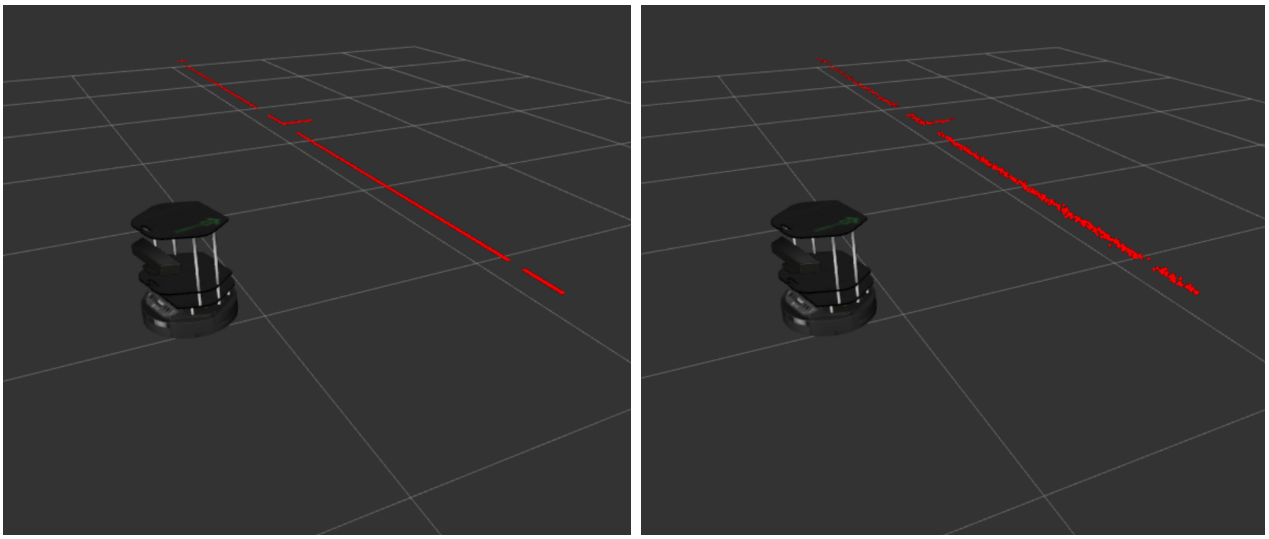
2.1.11 odom_noise

This node has three main purposes :

- Firstly, it has to get the pose of the robot published by Gazebo (via the `/gazebo/model_states` topic) to generate a ground truth odometry (we publish it on the `/GTOdom topic`). The existing `/odom` topic could be used, but it seems to be already noised (though we could not find it precisely in the documentations), and therefore, not very adapted.
- Secondly it noises this ground truth odometry and publishes it on the `/noisedOdom` topic. The details will be explained later.
- Finally, it publishes the different tf transforms needed by `gmapping` and `move_base` related to the noised and ground truth odometries.

2.1.12 laser_noise

It takes the values of the laser scans (on the `/scan` topic) and generates a noised version of these scans (published on the `/noisedScan` topic) (see Figure 8). Again details will be provided later.



(a) (b)

Figure 8: Without (8a) and with noise (8b)

2.1.13 life_cycle

This node spawns the harbour containers at the beginning of the simulation, and makes it appear and disappear through time. The goal was to imitate the behaviour of an harbour filled and emptied with a week period. Therefore there are two main alternating phases. The first one makes the containers disappear randomly, the second makes them reappear.

To be as close as possible to a real harbour, it was important to have quite a lot of containers (between 50 and 350, which leads to the need to automatise the process.

2.1.14 automove

This node gives waypoints to reach in order to circle in the "harbour". The idea was to move in straight lines in order to avoid navigation issues that can appear when turning too close around containers (see Figure 9).

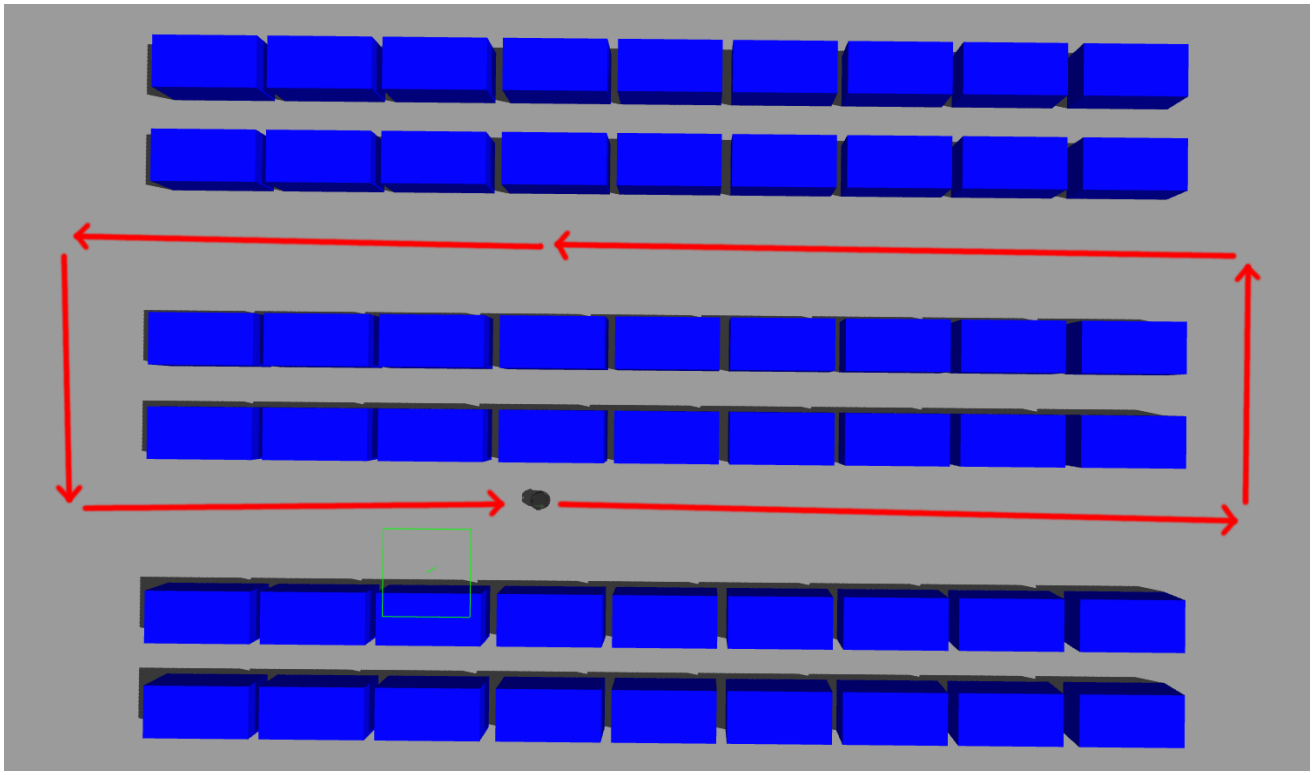


Figure 9: Waypoints used during the experiments

2.1.15 maps_saver

It regularly saves the map in images.

2.2 Details on the noise models

The goal here is to add a small error on the odometry and laser scan measurements in order to reproduce what may happen in the reality. If no noise was introduced in the simulation, it would be quite obvious that the localisation and mapping would be perfect, and therefore no derive would occur.

2.2.1 Odometry noise

Between two different instants, let $(\bar{x}, \bar{y}, \bar{\theta})$ be the initial pose, and $(\bar{x}', \bar{y}', \bar{\theta}')$ the final one (see figure 10). Then we can define :

$$\delta_{trans} = \sqrt{(\bar{x}' - \bar{x})^2 + (\bar{y}' - \bar{y})^2}$$

$$\delta_{rot1} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$$

$$\delta_{rot2} = \bar{\theta}' - \bar{\theta} - \delta_{rot1}$$

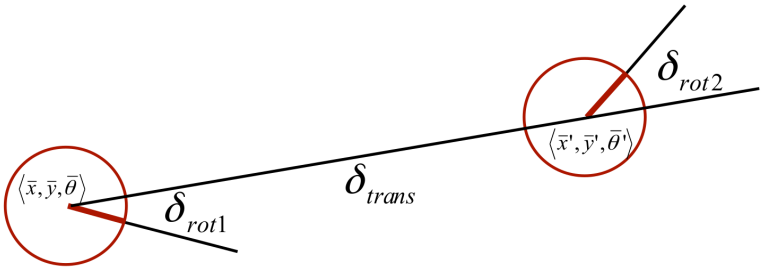


Figure 10: Odometry model [10]

Thus, the information carried by the odometry is reduced to $u = (\delta_{trans}, \delta_{rot1}, \delta_{rot2})$. The actual mesured odometry is corrupted with some noise and can be writen as [10] :

$$\hat{\delta}_{rot1} = \delta_{rot1} + \varepsilon_{\alpha_1|\delta_{rot1}|+\alpha_2|\delta_{trans}|}$$

$$\hat{\delta}_{rot2} = \delta_{rot2} + \varepsilon_{\alpha_1|\delta_{rot2}|+\alpha_2|\delta_{trans}|}$$

$$\hat{\delta}_{trans} = \delta_{trans} + \varepsilon_{\alpha_3|\delta_{trans}|+\alpha_4|\delta_{rot1}+\delta_{rot2}|}$$

Where ε_σ represents a random number following a normal centered normal law with a standard deviation σ. Finally, the noised pose can be obtained by :

$$x' = x + \hat{\delta}_{trans} \cos(\theta + \hat{\delta}_{rot1})$$

$$y' = y + \hat{\delta}_{trans} \sin(\theta + \hat{\delta}_{rot1})$$

$$\theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$$

Therefore, this noise model takes into account the magnitude of the movement of the robot, and is tunable with the α parameters. We can notice that the twist part of the odometry message is not impacted, though some other odometry noise models could.

2.2.2 Laser noise

The laser scan messages are composed of distance measurements for angles mapped from the minimum angle to the maximum angle, with a fixed step. It is easy to noise the distances by just adding a small value.

In order to know the magnitude of the noise, we took the characteristics of the Hokuyo URG-04LX. The noise is represented by a random number following a centered normal law where the standard deviation is :

- $\sigma = 0.01$ if $d \leqslant 1m$
- $\sigma = 0.01d$ if $d > 1m$

This 0.01 is the value described by the Hokuyo documentation, however, we let it as a parameter in order to play with it during our experiments.

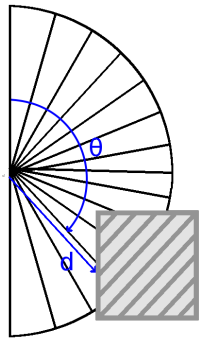


Figure 11: Laser model

2.3 Details on the sources

2.3.1 odom_noise_node.cpp

This is the source code for the `odom_noise` node.

Subscribed topics

- `/odom` : odometry provided by the Turtlebot
- `/gazebo/model_states` : ground truth pose provided by Gazebo

Published topics

- `/noisedOdom` : the noised odometry
- `/groundTruthOdom` : the ground truth odometry
- `/diffOdom` : the pose difference between the ground truth and noised odom, in an odometry message

Published tf transform

- `odom` → `GTOdom`

Parameters

- `alpha1`, ..., `alpha4` : odometry noise parameters

Functions

- `randomGaussianDouble` : generates a random number following a normal law (mu, sigma)
- `computeOdometryDifference` : computes the pose difference between two odometry messages
- `states_callback` : callback for `/gazebo/model_states`, fills the `GTOdom` global variable (*e.g.* ground truth odom). At the beginning, it initialises the ground truth and noised odometries to be equal.
- `odom_callback` : callback for `/odom`, fills the `lastOdom` global variable. At the beginning of the simulation, it computes the `initialOdomOffset` global variable that is used for tf transformations computations. Indeed, the initial pose given by `odom` and `GTOdom` is different since represented in different frames.
- `computeNoisedOdom` : this is the function that makes almost all the computations.
  It compares `GTOdom` and `lastGTOdom` (the ground truth odometry used during the previous call of this function) to generate the noises described in the part 2.2.1 and add it to `lastNoisedOdom` (the last noised odometry). Then, it publishes the different messages.
- `publishTF` : it computes and publishes the different tf transforms. The whole is enclosed in a `try...catch` block to prevent errors at the beginning, when the tf broadcaster is not fully initialised.

We discovered that the odometry provided by default in the `odom` topic was slightly noised, although it is not written anywhere in the documentations. It is why we decided to base our noised odometry on a ground truth odometry generated from Gazebo instead.

Therefore, we wanted to provide `move_base` and `gmapping` with a `noisedOdom` $\rightarrow$ `base_footprint` transform, meaning that the odometry frame is not anymore `odom`, but `noisedOdom`. However, while the noised odometry messages seems to be what we expected, the localisation and navigation with that is very bad. The problem probably comes from the tf transforms that we implemented at the end of the project, but we didn't found out exactly why.

In some of our experiments, we provided `move_base` and `gmapping` with the `noisedOdom` topic, but the `odom` frame. The behaviour of the robot was what could be expected if the noised odometry worked well. However, the results have to be considered with caution.

The most reliable results are when using the `/odom` topic and the `odom` frame instead, but it lacks the influence that could add a noised odometry.

The main assumption for going further on our experiments was that the odometry noise was not that important considering that the localisation is provided by gmapping, which relies a lot on the laser scans (for the localisation part at least). Moreover, in the real application, the map is supposed to have already been created with a very fine localisation system (RTK GPS). On top of that, the map drift process is likely to be mainly caused by bad scans when encountering a container newly spawned.

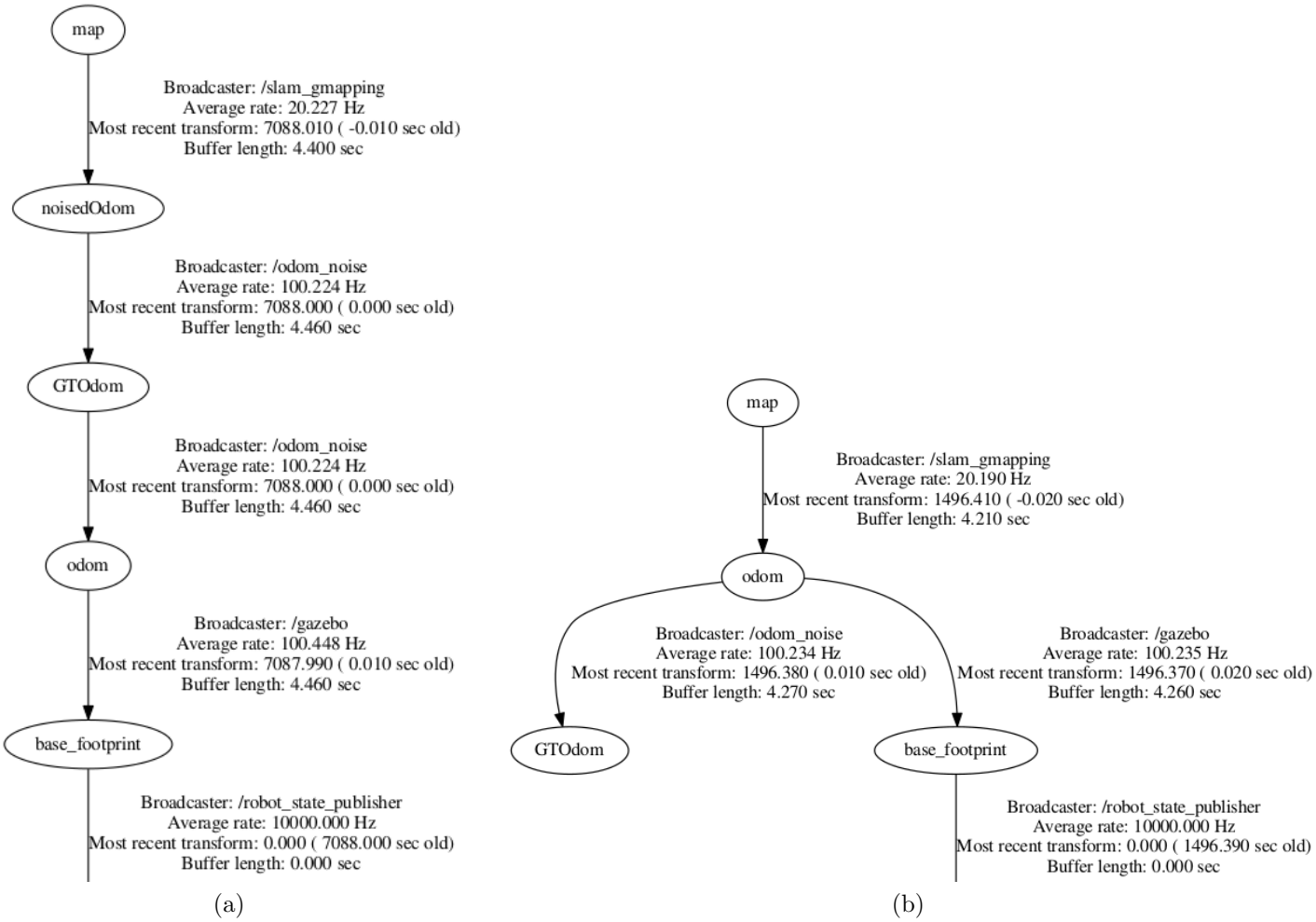


Figure 12: Frames based on noised odometry (12a) and on default odometry (12b)

2.3.2 laser_noise_node.cpp

This is the source code for the `laser_noise` node.

Subscribed topic

- `/scan` : the laser scan measurements

Published topics

- `/noisedScan` : the noised scan
- `/diffScan` : the difference between the real and noised scan

Parameter

- `sigma` : laser noise parameter

Functions

- `randomGaussianDouble` : generates a random number following a normal law (μ=a, σ=b)
- `scan_callback` : gets the scan measurement, noises it and publishes the different messages

By default on the simulation of the Turtlebot under Gazebo, the laser scan messages are generated from depth images taken by the Kinect by keeping only a horizontal line. The measurements given by it doesn't seem to be noised, as we can see in figure 8.

2.3.3 maps_saver_node.cpp

This is the source code for the `maps_saver` node.

Parameters

- `delay` : the delay (in seconds) between each save
- `path` : the absolute path to the directory where to save the images. If the directory doesn't exist, it will be created.

If there is a ros service to save the map, it isn't implemented for roscpp yet. This is why we where forced to call this service via a bash command inside the C++ code. Though non aesthetic, *it makes the coffee* as we say in French!

2.3.4 Container.h

Contains the description of a class Container sued to describes, well... a container.

It has the obvious arguments (`m_x`, `m_y`, `m_z`, `m_name`), a constructor and accessors. It also contains a friend function which is an overload of the `<<` operator from `std`.

2.3.5 Container.cpp

Defines the functions of the class `Container`.

2.3.6 detect_containers.cpp

Used to detect pair of dots in an image that are used to mark the presence of a container.

When trying to build our first testing environments, we realized how long it was taking and how unpractical it was to build these map by hand. Furthermore, we will have to make containers appear/disappear from the map during the testing, so we needed a program that could list all the containers we had placed on every testing environment (or gives this list ourselves!). Let's remember that our testing environments could contain from 50 to 350 containers during this project. It was obvious we had to find a way to optimise and automatise the environment creating process.

The choice made was to detect containers from a picture. A reliable way was to point the containers of the image with a dot, and a second dot next to it to give the orientation (horizontal [vertical] two-dots line for a horizontal [vertical] container). This method as the advantage to let one construct vast aisles of containers very easily. To make things easier, the dots are always situated on the top left corner of a container. Placing two dots in an oblique way will result in a container oriented at 45°.

Arguments

- `listCont` : list of the containers on the initialised map
- `nbContainers` : number of containers

These arguments are passed by reference to be modified.

First, the image is loaded in colours in a matrix thanks to opencv's `imread`.

Then we go through every pixel. If the current pixel inspected is coloured, we initialize a process that tries to verify it has a neighbour (if yes, then it's a container, if no it's just an error made while pointing the dots, or an already detected container). Since the pixels detecting a container are always on top left of it, we only need to check the 4 neighbours "under" the current one (it is also a way to ensure a container will not be detected 2 times by each of its two dots).

The second detected dot certifies of the detection of a container, and also allows to determines its yaw. A new object `Container` is thus created and added to `listCont`, while `nbContainers` is incremented.

The last thing to be explained is how the coordinates in pixels is converted to coordinates in Gazebo's coordinates. We multiply it by 2 ratios `img_scale` and biased scale. `img_scale` accounts for the pixels to meters ratio in the image. This ratio alone was enough for the scaling before we took into account the fact that our urdf container model was shorter than containers we were spotting on a photographs.

PLEASE NOTE that decomposing this in 2 ratios is only done to ease the understanding of why this works, since `img_scale` is actually present at the denominator of `biased_scale`. The only ratio left to worry about is actually the ratio:

$$\frac{\text{Length container in urdf model}}{\text{Nbr of pixels of a container in the loaded picture}}$$

which is fairly easy to tune. Feel free to replace those 2 ratios by the single ratio above if you feels like it.

2.3.7 detect_containers.h

The header of *detect_containers.cpp*.

2.3.8 life_cycle.cpp

This is our main script for shaping the testing environment. Its role is to initialize the map and spawn/delete containers at a given rate and with a given behaviour during the simulation. We also check that the containers spawn correctly (not on the robot for instance).

Some global variables are defined in the beginning, among which 5 of them are set as parameters in the launch file *world.launch*:

Parameters

- `maplimits` : defines the limits of the virtual map inside which a container can be spawned. This parameter is only used for the random spawn mode, and not for the (re)spawn mode which we usually use in our simulations. Expressed in gazebo's meters.
- `minsleeptime` : minimal waiting time between two actions of the life-cycle (an "action" being either spawning or deleting a container). Expressed in seconds.
- `randsleeptime` : upper bound of the additional random waiting time between two actions. Expressed in seconds.
- `leftoversratio` : Between 0.0 and 1.0. proportion of the harbour filled, below which it's considered empty (or full considering $(1 - \texttt{leftoversratio})$) and will try to change it's action mode.
- `securitySpawningDistance` : Security distance between the container and the robot during spawning, in meters.

Subscribed topic

- `/groundTruthOdom` : associated with a callback function `callback_odomReceived`. Its role is to receive the absolute position of the robot and to stock it in variables `robotPose_x` and `robotPose_y`.

Services called

- `gazebo_msgs/SpawnModel` : for spawning a container
- `gazebo_msgs/DeleteModel` : for deleting one.

The urdf model is loaded and parsed. It is then loaded into a `gazebo_msgs::SpawnModel` object. The list of containers detected is generated by calling `detect_containers` and the map is generated by spawning these initial containers. A pause is done to let the operator launch the various launchfiles needed for the test, and to let the robot explore the map a little before containers start to disappear.

The main loop consists of an alternation of `actionmode`=1 (deleting mode) and `actionmode`=0 (spawning mode). This alternation is made with a delay of time `sleeptime` and consists of an alternation of phases.

The most simplistic phase is just a random choice of the mode, but we computed a more interesting way of doing it for the sake of our simulations. The mode doesn't try to change until the harbour is considered empty (under the `leftoversRatio` for `actionmode`=1) or full (above (1 − `leftoversRatio` for `actionmode`=0). When it tries to change mode, the probability that it does it is inversely proportional to the number of containers left (resp. the number of containers not already respawned). This probability can be biased with the `clearingFactor` (increasing it means you have more chance to stay in delete mode until there are no containers left).

The spawning mode was initially spawning containers randomly on the map, which wasn't really useful in simulations. We changed it so that the containers deleted are stocked in a list, and the spawning mode consist in respawning containers at their old emplacement.

2.4 Details on the launch files and parameters

As we saw, there are many nodes to launch in order to make the experiment work, and also lots of parameters to tune to adapt the behaviour of `gmapping` and `move_base`. The two next launch files are enough to start everything, but it includes and relies on other launch and config files.

2.4.1 world.launch

The first launchfile to launch. This one launches Gazebo with an empty world, then spawns the Turtlebot with the laser, and launch the `life_cycle` node.

Among the global arguments, we can choose the sensor used to get the laser scans ("kinect" or "hokuyo"). The Hokuyo solution has been implemented at the end of the project, but is not fully working yet. However, it will be a better choice in the future : one of the problem we encountered was that the angle of the laser scans with the Kinect can't be higher than 180°(because of its pinhole camera model). This limits what the robot detects, which drives to lower quality maps. For instance in some of our experiments, while the robot was between two rows of containers, it was able to detect only one. The Hokuyo telemeter has not this problem, with a higher angle detection (up to 270°).

The node `life_cycle` needs few arguments to work:

- `mapImagePath` : the path to an image representing where the containers must spawn. It can be absolute, or relative to the root of the package.
- `maplimits` : the size of the map
- `minsleeptime` : minimum time between two container spawns/deletes (in seconds)
- `randsleeptime` : upper bound of the additional random waiting time between two actions (in seconds)
- `leftoverRatio` : proportion of the harbour filled, below which it's considered empty (or full considering (1- leftoversratio)) and will try to change it's action mode

- `securitySpawnDistance` : the minimum distance to the Turtlebot to accept a container spawn. It avoids creating containers on the robot.

2.4.2 navigation.launch

The second launchfile to launch. This one handles the robot behaviour by launching `gmapping`, the noise nodes, `move_base`, `map_saver`, and `automove`. Let's look at the main arguments needed by these nodes.

odom_noise

- `alpha1`, ..., `alpha4` : the odometry noise parameters (cf 2.2.1)

laser_noise

- `sigma` : the variance of the gaussian noise on the laser measurements (in meters) (cf 2.2.2)

gmapping

- `scan_topic` : the topic of the laser scans. It can be either `scan` or `noisedScan`-
- `base_frame` : the base frame of the Turtlebot. Not to be changed.
- `odom_frame` : the odometry frame. It can be `odom` or `noisedOdom`.

move_base

- `odom_frame_id` : the odometry frame. It can be `odom` or `noisedOdom`.
- `base_frame_id` : the base frame of the Turtebot. Not to be changed.
- `laser_topic` : can be `scan` or `noisedScan`
- `odom_topic` : can be `odom` or `noisedOdom`

map_saver

- `delay` : the delay (in seconds) between each save
- `path` : the absolute path to the directory where to save the images. If the directory doesn't exist, it will be created.

The following files are used by the preivous launch files. These are mostly modified versions taken from the `turtlebot_navigation` package in order to tune the parameters, or files used to enable the Hokuyo.

2.4.3 gmapping.launch.xml

There are many parameters already tuned for the Turtlebot, but we can still play on them to tweak the impact of the laser for instance.

- `sigma` : its purpose is not very clear, but we noticed than turning it to zero prevented the robot to be constantly teleported on the map.
- `srr, srt, str, stt` : odometry errors parameters. Turning it to zero means that gmapping totally trusts the odometry. Its not necessary something we want, but since the odometry is considered quite good on a Turtlebot, we have to put small values.
- `maxRange` : maximum range of the sensor
- `maxUrange` : maximum usable range of the sensor. The measurements are cropped to this value.

We can play on these two last values, and on kinect/hokuyo parameters to increase the range of the sensors in order to see farther.

2.4.4 move_base.launch.xml

The only modifications we made there were to modify the parameters file included to ours. These files mostly defines the behaviour of the local and global planners.

The behaviour of the robot when passing close to obstacles, the want to change robots parameters such as its size, etc... are many reasons why we thought using our own parameters yaml files for the costmaps was a good idea. The three next yaml files are important for costmaps tuning.

2.4.5 costmap_common_params.yaml

Defines parameters commons to both costmaps, such as the robot's shape.

- `robot_radius` : for a circular robot, the radius of its circular footprint. We tried to set it higher than the real radius sometimes to force the robot to stay in the middle of the aisles. It's hard to tune this way since it can cause the robot to be stuck easily.
- `track_unknown_space` : *true* needed for enabling global path planning through unknown space
- `inflation_layer:enabled` : *true*, inflates obstacles to avoid getting to close
- `inflation_layer:cost_scaling_factor` : the exponential rate at which the obstacle cost drops off. We sometimes tried to decrease it because the robot happened to be stuck next to a container so we tried to restrain it from passing too close to obstacles.

- `inflation_layer:inflation_radius` : max distance from an obstacle at which costs are incurred for planning. Setting it bigger restrains from passing too close to obstacles, but makes it harder to plan optimal paths.
- `static_layer:enabled` : *false*. See more in part 3.1.

2.4.6 global_costmap_params.yaml

Defines parameters for the global costmap.

- `update_frequency` and `publish_frequency` : setting these higher allows better theorical performances, but is more computationally complex.
- `static_map`=*false* and `rolling_window`=*true* : also more on this in the part 3.1. A `static_map` mode requires to give a static costmap, useful for example if we want to test the influence of non-moving part of the map (refrigeration units...).
- `width` and `height` : determine the size of the costmap. With large environment, a big costmap is required for an effective path planning, but that requires more computational time.

2.4.7 local_costmap_params.yaml

Defines parameters for the local costmap.

- `update_frequency` and `publish_frequency` : generally set higher to the ones of the global costmap because the local planning is crucial to avoid collisions (especially in our project). It isn't really a problem since the global costmap is way smaller.
- `width` and `height` : the size of the costmap must be suited to the robots size, to the laser sensor range and also to the speeds of the robot.

A more extensive and complete explanation on how to use these yaml files are given in [11].

The next files are modified to, on the one hand add the Hokuyo to the Turtlebot, and on the other, modify the behaviour of the Kinect. Many modifications concern the change of links between files.

"Kobuki" represents the version of the Turtlebot and "hexagon" the form of the basis.

2.4.8 kobuki.launch.xml

This launch file is the first one launched after calling *world.launch*. It will call the urdf description of the robot in its two different versions : with Kinect or with Hokuyo.

2.4.9 kobuki_hexagons_kinect.urdf.xacro and kobuki_hexagons_hokuyo.urdf.xacro

Describes the location of the sensors on the Turtlebot, and also includes the next file.

2.4.10 turtlebot_gazebo.urdf.xacro

As it configurates the Kinect and Hokuyo specifications in the Gazebo simulation, this is the true file to modifiy in order to change the behaviour of the sensors.

We can change there the range of the Kinect (in the `<clip><far> </far></clip>` tag) and its angle (in `<horizontal_fov> </horizontal_fov>`). However, this angle can't be as high as we wanted, even if we increased it from its initial value since the Kinect relies on a pinhole camera model. This is the reason why using the Hokuyo laser sensor could be a better idea.

In the Hokuyo definition, the node handling the sensor is launched as a Gazebo plugin. We found different versions to launch it, but neither did. Thus, the problem preventing the Hokuyo to work probably lays here.

2.5 Installation instructions and reproduction of the experiment

These are the steps that we followed to have the experiment ready. Of course, it didn't work well the first time and we needed to install new packages in the process, or reinstall different versions. This suppose ROS is already installed in the computer.

We worked on ROS Kinetic and Ubuntu 16.04, so there should be some adaptations to do for other distributions. Catkin is the workspace manager used.

2.5.1 Installation of Turtlebot [12]

```
1  sudo apt-get install linux-headers-generic
2  sudo sh -c 'echo "deb-src http://us.archive.ubuntu.com/ubuntu/ xenial main
       restricted deb-src http://us.archive.ubuntu.com/ubuntu/ xenial-
     updates main restricted deb-src http://us.archive.ubuntu.com/ubuntu/
     xenial-backports main restricted universe multiverse deb-src http://
     security.ubuntu.com/ubuntu xenial-security main restricted" > \ /etc/
     apt/sources.list.d/official-source-repositories.list'
3  sudo sh -c 'echo "deb-src http://us.archive.ubuntu.com/ubuntu/ xenial main
       restricted
4  deb-src http://us.archive.ubuntu.com/ubuntu/ xenial-updates main
     restricted
5  deb-src http://us.archive.ubuntu.com/ubuntu/ xenial-backports main
     restricted universe multiverse
6  deb-src http://security.ubuntu.com/ubuntu xenial-security main restricted"
       > \ /etc/apt/sources.list.d/official-source-repositories.list'
7  sudo apt-get update
8  sudo apt-get install ros-kinetic-librealsense
9  sudo apt-get install ros-kinetic-turtlebot
```

2.5.2 Installation of the latest version of Gazebo 7 [13]

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get install gazebo7
sudo apt-get install libgazebo7-dev
```

2.5.3 Installation of Turtlebot Gazebo [14]

```
sudo apt-get update
sudo apt-get install ros-kinetic-turtlebot-simulator
```

2.5.4 Installation of urg_node (for the Hokuyo

```
sudo apt-get update
sudo apt-get install rros-kinetic-urg-node
```

2.5.5 Installation of our package

```
cd <ROS catkin workspace>
source devel/setup.bash
cd src
git clone https://github.com/CorentinChauvin/evolutive_map
catkin build  # (or any other catkin build tool)
```

2.5.6 Starting the experiment

In three separate consoles (don't forget to source the *devel/setup.bash* file) :

```
roslaunch evolutive_map world.launch  # (wait then for the containers to be spawned)
roslaunch evolutive_map navigation.launch
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

2.5.7 Creation of a containers map

If you want to point containers on an existing picture, a good way is to use a software allowing to draw over calques, such as Paint.NET. Import the picture as base, create a new calque over it and start pointing the containers' top left corner with two dots as described on the picture below (Figure 13). Remember that the lengths and width of containers can vary, as well as the picture scale. As described earlier, the only ratio you have to tune is the one in *detect_containers.cpp*:

$$\frac{\text{Length container in urdf model}}{\text{Nbr of pixels of a container in the loaded picture}}$$

Note that the length/width ratio can vary too so be sure to not point containers that may be too close, so that they doesn't overlap once converted in urdf models at gazebo's coordinates.

If you just want to make a map as you imagine it, any picture editor/creator such as Paint will do. Draw two dots in the desired direction, copy them and paste them at least m pixels

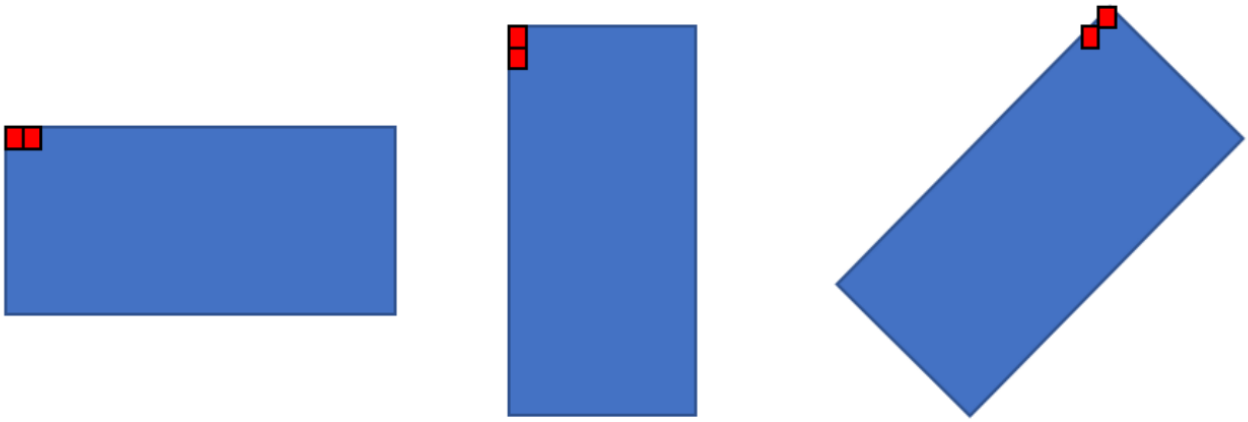


Figure 13: Container pointing principle

from the first ones (where m = nbr of pixels of a container in the picture) and repeat. You can accelerate the process by copying a larger group of pixels from time to time, and you'll have an entire map in less than a minute!

2.5.8 Generate a static map

The easiest way is to first generate the dynamic environment, and let it spawn on gazebo. Then spot the coordinates where there is space to put the static elements. Note those coordinates, launch gazebo to have a new empty map and plant the static elements at the desired coordinates. We give below, for instance, a list of possible coordinates to put a static object (urdf model: "Lamp Post") on the map *St_Nazaire_EXP.jpg*, and the name of the container the closest to it :

```
1  Closest container coordinates: Box213 (-10.6, -5.7) -> possible placement
       of a static object: (-10.7, -7.0)
2  Closest container coordinates: Box155 (-10.4, 5.0) -> possible placement
       of a static object: (-10.5, 3.7)
3  Closest container coordinates: Box97 (-10.2, 15.6 )-> possible placement
       of a static object: (-10.3, 14.3)
4  Closest container coordinates: Box39 (-10.1, 26.0 )-> possible placement
       of a static object: (-10.2, 24.7)
5  Closest container coordinates: Box222 (9.4, -5.7) -> possible placement of
        a static object: (9.3, -7.0)
6  Closest container coordinates: Box164 (9.6, 5.0 )-> possible placement of
       a static object: (9.5, 3.7)
7  Closest container coordinates: Box106 (9.8, 15.6) -> possible placement of
        a static object: (9.7, 14.3)
8  Closest container coordinates: Box48 (9.9, 26.0) -> possible placement of
       a static object: (9.8, 24.7)
```

You can then export this as a world file. A good thing is to delete the Turtlebot elements before saving the map, but if you forget to do so you may have issues when it tries to spawn a

Turtlebot later on your map, but gazebo will find usually a way around it.

Launch the Turtlebot teleop node, as well as gmapping and rviz and navigate the Turtlebot around the map manually. Make sure every static element is fully detected from every angle.

If you have closed Gazebo, you need to relaunch your map (and the Turtlebot). You can modify *launch_empty_gazebo.launch* by changing the argument `world_file` to yours, then do:

```
roslaunch evolutive_map launch_empty_gazebo.launch

# And then:
roslaunch turtlebot_navigation gmapping_laser.launch
roslaunch turtlebot_rviz_launchers view_navigation.launch
roslaunch turtlebot_teleop keyboard_teleop.launch

# Once the map is done:
rosrun map_server map_saver -f [path where to save the static_costmap]
```

Your static costmap is ready to be used.

3 Issues encountered and simulation results

3.1 Main issues encountered

During this project, we faced many problems and bugs. We managed to solve some, but not all of them.

The first of them was the combined use of `move_base` and the `keyboard_teleop` tool. The latter allows us to move the Turtlebot thanks to the keyboard. However, if the corresponding node works, the velocity commands sent by `move_base` have no effect at all.

Then, by default, the Turtlebot can't navigate in unknown space (*e.g.* space where the map is not built yet) for some safety reasons. Nevertheless, we wanted to change that in order to speed up the discovery of the map. Therefore, we needed to set to *true* the `track_unknown_space` param in *costmap_common_params.yaml*, and `allow_unknown` in *navfn_global_planner.yaml*.

Likewise, the `move_base` doesn't authorise a navigation off the global costmap, and this can not be changed. And often, at the beginning of the simulation, the global costmap was very small, or not well centered. A solution to that was to allow the global costmap to grow when the robot had to navigate outside it. This can be done by setting `rolling_window` to true and static_map to false in *global_costmap_params.yaml*, and `static_layer`: enabled to true in *costmap_common_params.yaml*.

Again about the global costmap, we witnessed weird trajectories when navigating in unknown space. Indeed, sometimes when the goal was very deep in the unknown, the trajectory generated by the global planner was to go along the border of the global costmap to go as close of the goal as possible, and reach it in straight line. Fortunately, this problem appeared only in early isolated tests, and not during the real experiments.

A real issue is the behaviour of the robot close to the obstacles. The global planner tries to give the shorter path to the goal, and if its parameters are not well tuned, it can force the robot to go close enough to the obstacles that it considers itself in trouble. This problem can be watched in two situations : when the Turtlebot has to pass between two obstacles in a narrow passage. Or when it wants to go to an other row of the "harbour" while discovering the map at the same time.

Tuning has a real impact, but it is very hard to find the good ones. A simple solution to avoid any trouble was to give a path trough the harbour which didn't needed to pass close to containers, with straight lines cleared of any obstacles (even if some appear or disappear during the process).

Sadly, the noised odometry is not ready yet. We initially thought that providing a `/noisedOdom` topic to `move_base` and gmapping was enough, but we discovered then that they also rely on tf transforms. Building the needed transforms are not something very easy, considering that we can't change the already existing `odom` $\rightarrow$ `base_footprint` transform. It is the reason why we built intermediate transforms. But the very bad localisation and mapping induced shows that we missed something in the process.

Likewise, we implemented the files needed for the Hokuyo to work at the end of the project, but somehow, the node handling the Hokuyo simulation is not getting launched. Cf 2.4.10 for some additional information.

Speeding up the simulation could be something very interesting to reproduce a lot of experiments and do comparisons. Since, the way the containers appear and disappear, just as the noise generated, are random, differences occure during two experiment with the same parameters. We can modify the speed of the Gazebo simulation by changing a parameter in the Gazebo world physics (`real time update rate`) or in the *empty.world* file (`real_time_factor`).

However, this leads to many issues dealing with the bad synchronisation of the different nodes at high speed, and produces big localisation errors even if the speed is not so high. On top of that, the simulation is so heavy that just multiplying the simulation speed by ten is already to much for the computers.

Finally, the biggest issue is the instability of the tools used, and mostly when Gazebo is concerned. Gazebo coupled to ROS, or Rviz, often have graphics issues at starting, and the only solution to that is to Ctrl^C the node and launch it again. These issues are well known, and there is no alternative but upgrading Gazebo to its latest version (which is not advised dealing with ROS Kinetic)(and not sure it is enough so much it is unstable).

On top of that, the Gazebo API for deleting models on the map works one time over three, and again, the only solution is to stop the experiment and start it again (fortunately, when it begins to work, it works during the whole experiment).

3.2 Simulations

The simulations we conducted in the end were based on the *ExperimentMap_medium.jpg* file (around 50-60 containers). By default, we have the following parameters: `minsleeptime` = 60,

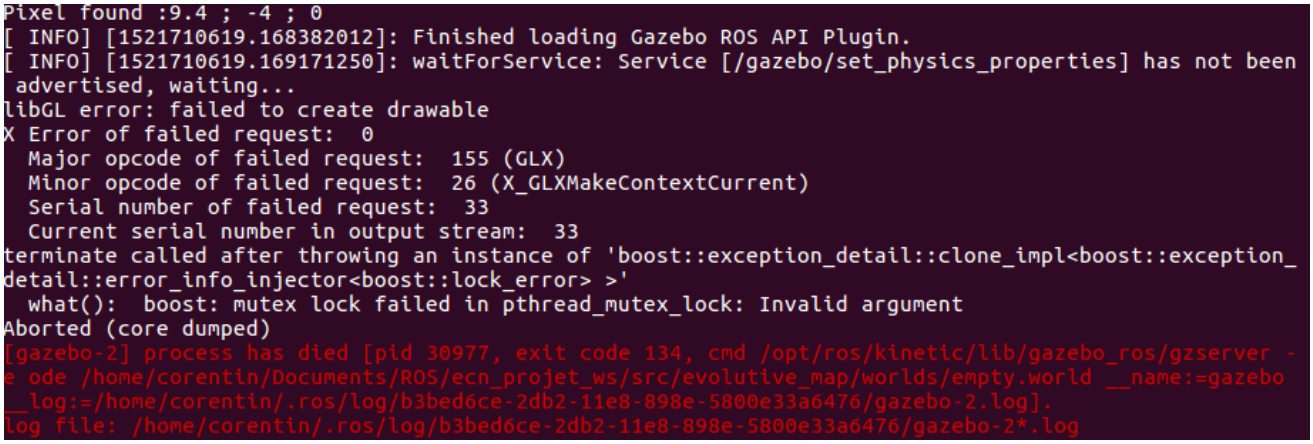


Figure 14: Gazebo graphics error while starting

`randsleeptime` = 120, `leftoversRatio` = 0.1, `clearingFactor` = 10.0, 120 seconds of wait-time between the end of the map initialisation and the beginning of the life-cycle of the map. The costmap parameters are visible in the file as they currently are, but we can at least points out that we use a rolling window for the global costmap.

The waypoints are : (13.8, 0); (13.8, 5); (0, 5); (-9, 5); (-9, 0); (0, 0). They make a rectangle passing by the two central aisles of containers (see Figure 9).

The `real_time_factor` is 1.

Aside from this report, you'll find some gif files animating the maps, which brings a better view of what's happening.

3.2.1 Experiment 1

Parameters of the simulation:

- No noise on odometry
- No noise on laser scans
- No evolution of the map
- Duration : 2 hours

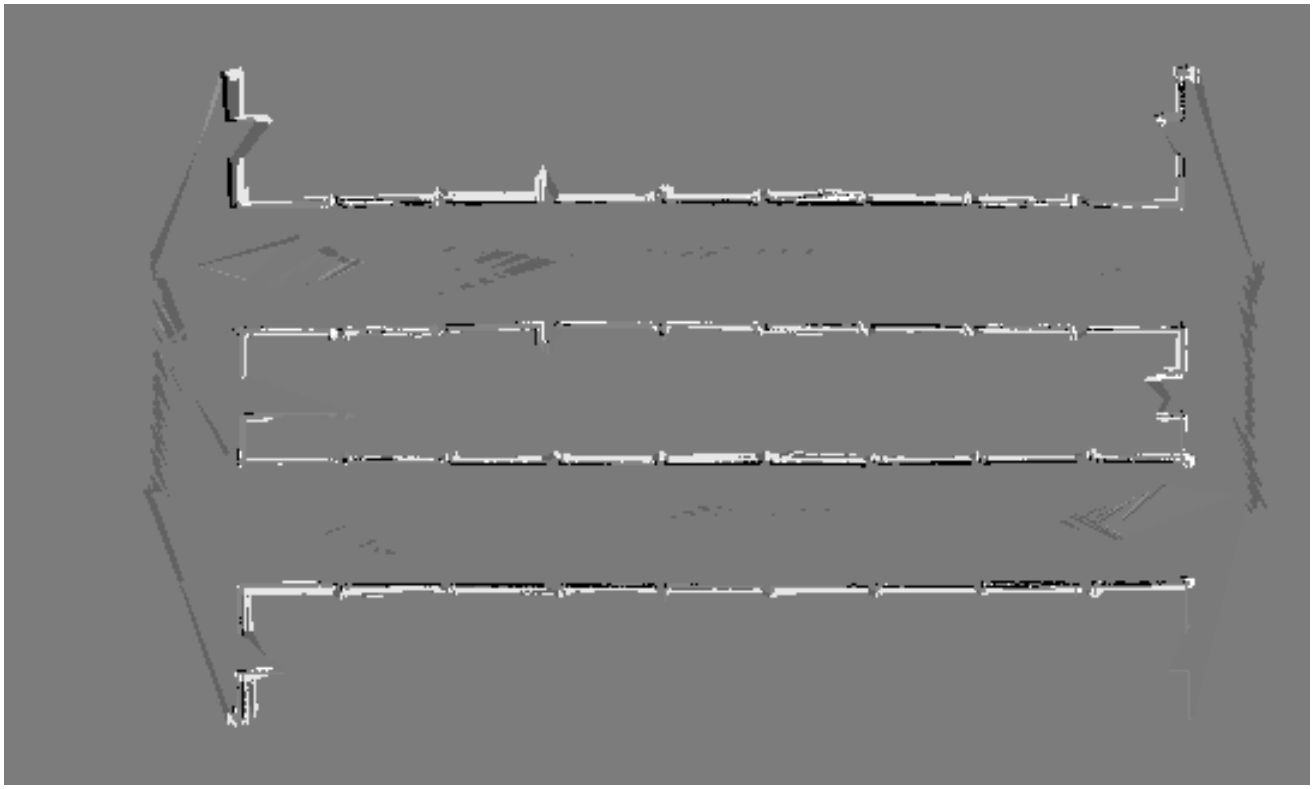


Figure 15: Experiment 1. In black, initial map. In white, final map.

We can see on figure 15 that without any noise and any map change, drift is not easily noticable. There is obviously tiny differences between the initial and final maps. However, it is hard to say whether it is due to a real drift, or more by small simulation noises (because of numerical errors).

3.2.2 Experiment 2

Parameters of the simulation:

- Noise on odometry (only on the `/odom` topic)
- Noise on laser scans
- Evolution of the map (first phase of disappearance of containers)
- Disappearance every 1 to 3 minutes
- Duration : 1 hour and a half

We see clearly on figure 16 the shift occuring during the experiment. The map rotates, making containers appearing on the map meters away their actual location.

During the experiment, as less and less containers are left, it becomes harder for the robot to localise itself well. At the end, the robot doesn't see any container for a long time, relying entirely on odometry, which is noised.

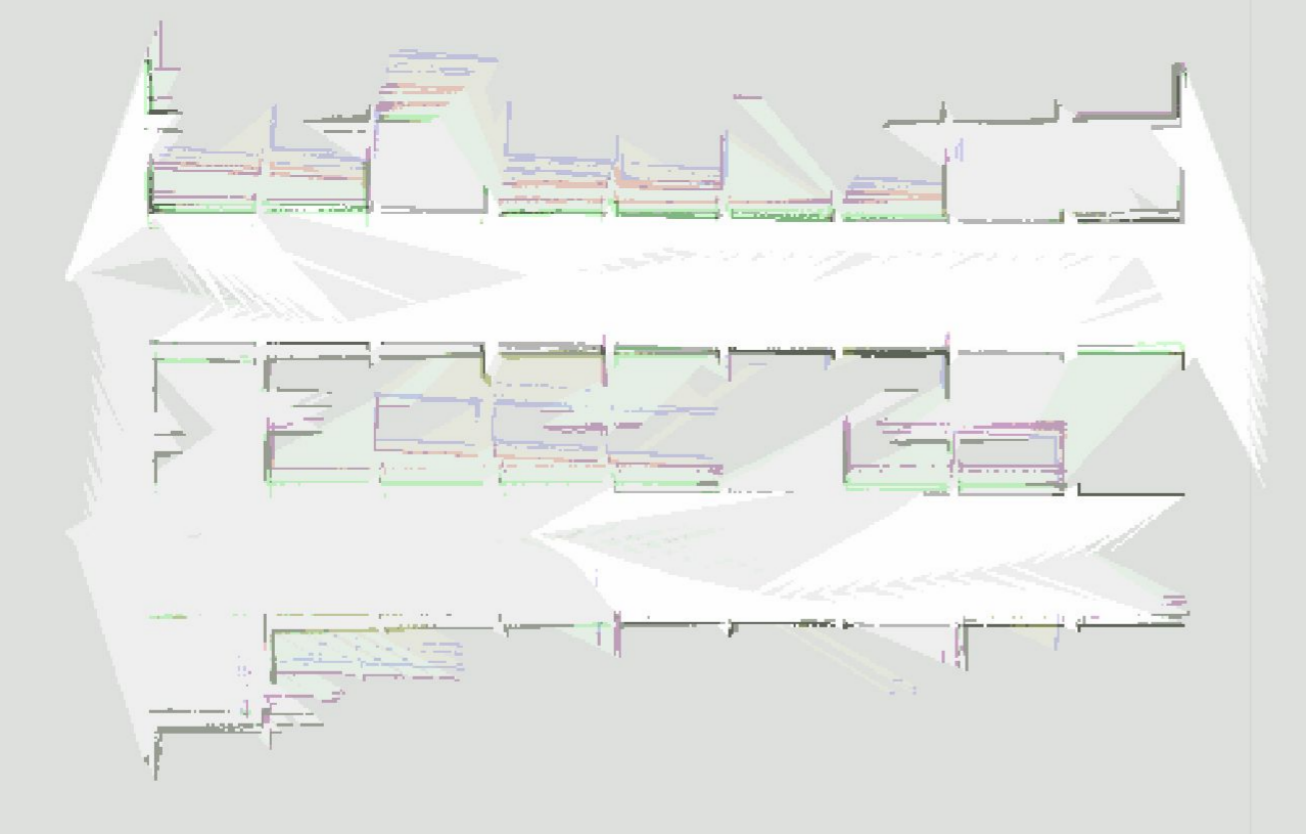


Figure 16: Experiment 2. Initial map in black, then in green, red and blue.

3.2.3 Experiment 3

Parameters of the simulation:

- Noise on odometry (only on the `/odom` topic)
- Noise on laser scans
- Evolution of the map (first phase of appearance of containers)
- Appearance every 1 to 3 minutes
- Duration : 1 hour and a half

This experiment is the prolongation of the previous one, starting from an environment where there are few containers left, and with a map that has already drifted. We see on figure 17 that the appearance of the containers, which brings more and more details to compare with the map,

is not enough to counteract the drift. As the map had already drifted at the beginning, the containers newly spawned are added at wrong positions on the map. Therefore, at the end of the experiment, the localisation is completely wrong.

Figure 17: Experiment 3

3.2.4 Experiment 4

Parameters of the simulation:

- No noise on odometry
- Noise on laser scans
- Evolution of the map during a full cycle
- Appearance every 1 to 3 minutes
- Duration : 2 hours

Without odometry noise, we can notice on figure 18 that the drift is way less important. There are differences between the initial and final maps, but it can either be because of a map drift, or the laser scan noises that blur the shape of the containers.

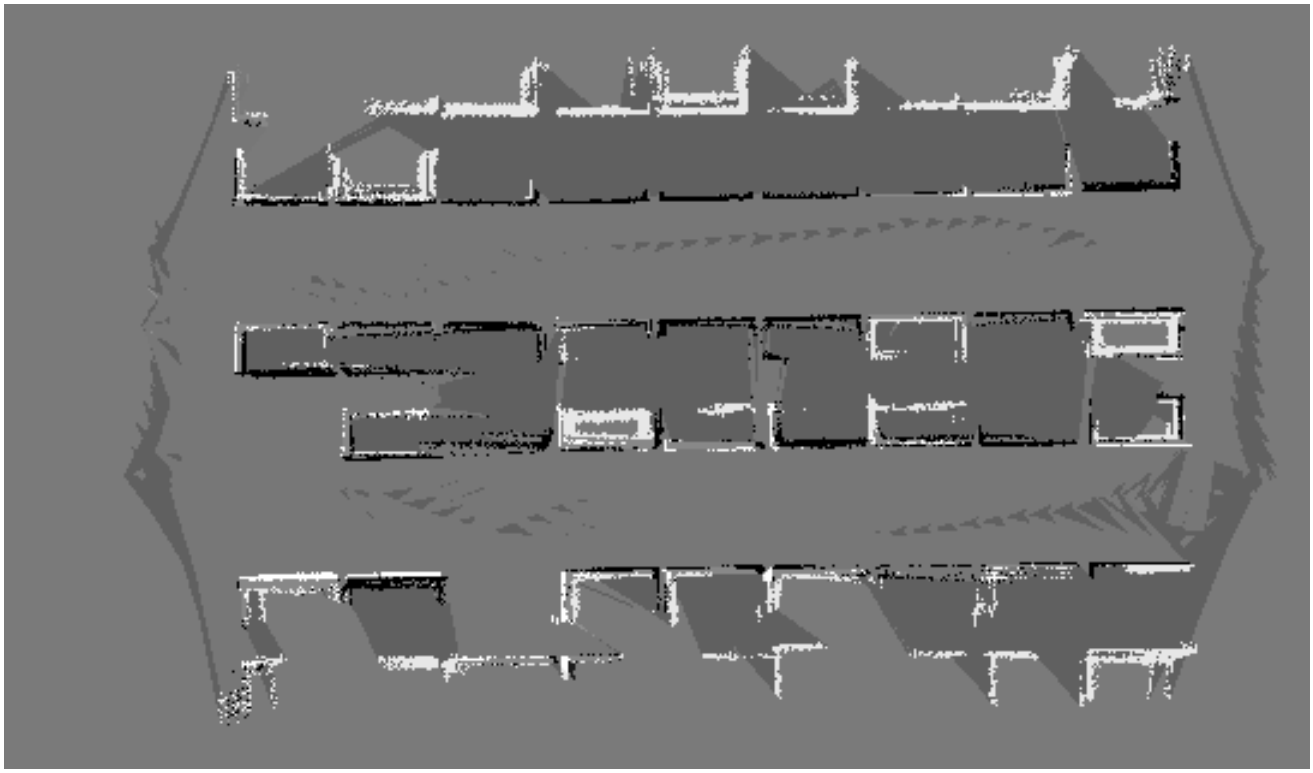


Figure 18: Experiment 4. In black, initial map. In white, final map.

3.2.5 Experiment 5

Parameters of the simulation:

- No noise on odometry
- Noise on laser scans (doubled compared to the previous experiment)
- Evolution of the map during a full cycle
- Appearance every 1 to 3 minutes
- Duration : 2 hours

The differences shown by figure 19 are clearer that for the previous experiment, but the conclusions said there are still valid.

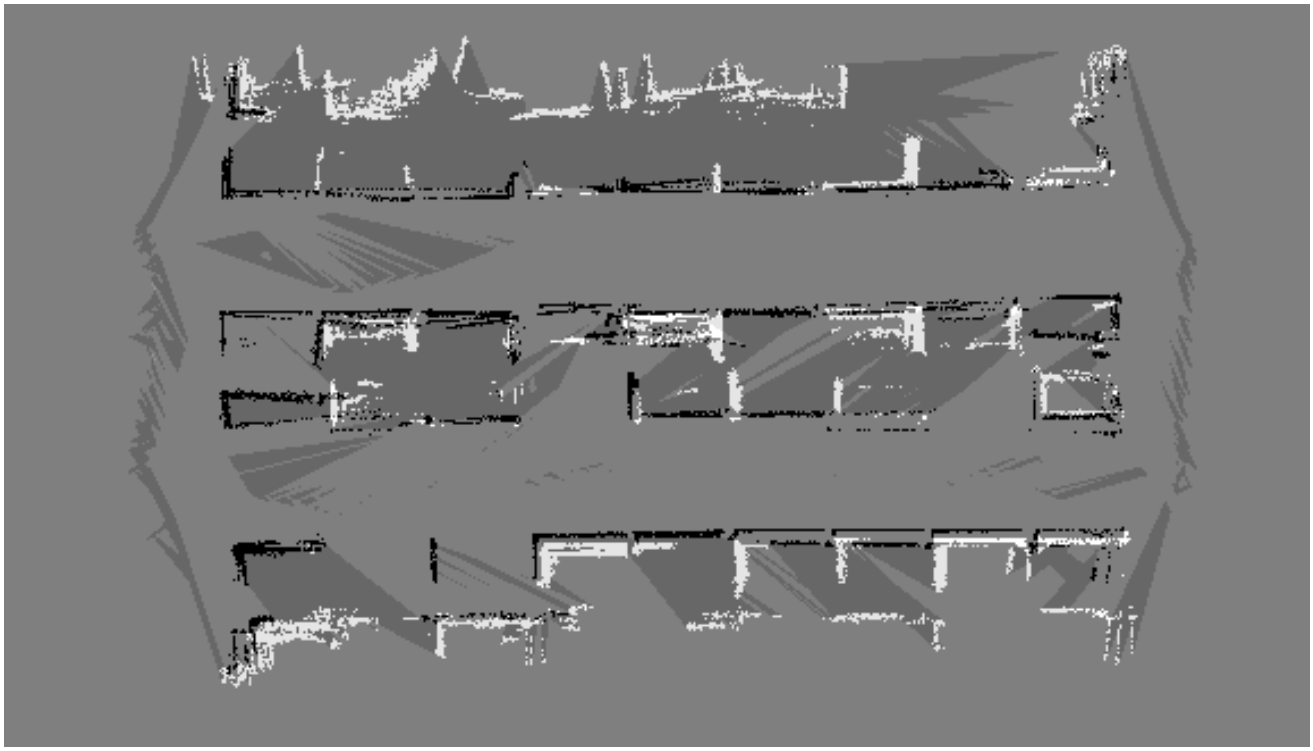


Figure 19: Experiment 5. In black, initial map. In white, final map.

3.3 Results analysis

We can first notice the bad bahaviour of gmapping when it comes to forget disappeared containers. Indeed, when a container is not present anymore but still in the map, not seeing it with the laser is not enough to delete it of the map. There must be a container behind so that the distance returned by the laser is not *NaN*. This is probably for safety reasons, where gmapping wants to be sure that their is nothing before deleting it of the map.

Moreover, even if gmapping wants to delete a container of the map, it makes it disappear progressively. The time needed for a container to be erased is way too long in our experiments, the Turtlebot must pass several times next to the missing container (representing minutes of simulation).

Aside that, the noises on laser scans seem to blur the shape of the containers during the experiment. Thus it is hard to quantify the impact of the drift.

The laser wide angle is an other problem. Since it is small, the robot has troubles to detect its environment. For instance when navigating between two rows of containers, it can only detect one. And as it can't detect behind itself, the spaces between the containers are not well detected. This leads to bad quality maps (see 2.4.1 for more details).

Finally, the results where odometry is noised has to be taken with caution. Indeed, we pro-

vided there `gmapping` and `move_base` the `noisedOdom` topic (that we think valid) coupled with the `odom` frame. We don't know how these nodes really use the topics and frames, and so we can't be sure it works the way we expect. The only thing we are sure is that noising the odometry topic impacts the mapping, and seems to make the map derives.

3.4 Perspectives

There are some improvements or further tests that are ready to be done but that we did not have the time to do.

1. Using a static map (static elements, for example refregiration units). We think it can anchor the map to known elements of fixed position, and thus prevent drift.
2. Use other noise parameters for odometry and laser scans.
3. Play with other parameters: rate of apparition/disappearance of containers, number of "cycles" of apparition/disappearing, repartition of the waypoints (random sampling?).

Besides, there are lots of other ways to improve the experiment, that are more or less already implemented in our package.

- Make the Hokuyo work, instead of using the Kinect, in order to have a greater angle for laser scans.
- Find and modify the "forgetting factor" of `gmapping`, and the fact that it can only forget an obstacle if the laser comes back from another obstacle behind (it'll probably need lots of work, even just to find where it is located in the source code).
- Use the many alternative SLAM methods in ROS: HectorSLAM, KartoSLAM, CoreSLAM, LagoSLAM...

References

[1] David Fillat. *Cartographie et localisation simultanées en robotique mobile*. Techniques de l'ingénieur, 2014.

[2] Patrick Durand, René Durand, Dzintars Avots, Edward Lim, Romain Thibeaux, and Sebastian Thrun. *A Probabilistic Technique for Simultaneous Localization and Door State Estimation with Mobile Robots in Dynamic Environments*. 2002.

[3] Wiki ROS. robot_state_publisher. `http://wiki.ros.org/robot_state_publisher`.

[4] Wiki ROS. tf transforms. `http://wiki.ros.org/tf`.

[5] Wiki ROS. kobuki_safety_controller. `http://wiki.ros.org/kobuki_safety_controller`.

[6] Wiki ROS. cmd_vel_mux. `http://wiki.ros.org/cmd_vel_mux`.

[7] Wiki ROS. depthimage_to_laserscan. `http://wiki.ros.org/depthimage_to_laserscan`.

[8] Wiki ROS. slam_gmapping. `http://wiki.ros.org/slam_gmapping`.

[9] Wiki ROS. move_base. `http://wiki.ros.org/move_base`.

[10] Wolfram Burgard, Cyrill Stachniss, Maren Bennewitz, and Kai Arras. *Introduction to Mobile Robotics, Probabilistic Motion Models*. 2011.

[11] Wiki ROS. Robot setup for navigation. `http://wiki.ros.org/navigation/RobotSetup`.

[12] ROS Answers. Turtlebot installation. `https://answers.ros.org/question/246015/installing-turtlebot-on-ros-kinetic/`.

[13] Wiki ROS. Turtlebot gazebo bringup guide. `http://wiki.ros.org/turtlebot_gazebo/Tutorials/indigo/Gazebo%20Bringup%20Guide`.

[14] Gazebo. Gazebo 7 installation. `http://gazebosim.org/tutorials?tut=install_ubuntu&ver=7.0#Alternativeinstallation:step-by-step`.

[15] Rahul Biswas, Benson Limketkai, Scott Sanner, and Sebastian Thrun. *Towards Object Mapping in Non-Stationary Environments With Mobile Robots*. 2002.

[16] Denis Wold and Gaurav S. Sukhatme. *Towards Object Mapping in Non-Stationary Environments With Mobile Robots*. 2004.

[17] Wolfram Burgard, Sebastian Thrun, and Dieter Fox. *Probabilistic robotics*. 2000.