Mobile Robots for Critical Mission: Final Project

Submitted by:

Francesco Rosa (8802000026)

UNIVERSITÀ DEGLI STUDI
DI SALERNO

Department of Computer Engineering, Electrical Engineering and Applied Mathematics

University of Salerno

July 2022

Contents

1 Introduction **i**

1.1 Problem Description i

1.2 Possible Solutions ii

2 Proposed Solution **iv**

2.1 Localization: Proposed Solution iv

2.1.1 Extended Kalman Filter iv

2.1.2 Adaptive Monte-Carlo Localization viii

2.1.3 Bring-up: Automatic Localization Procedure xi

2.2 Planning: Proposed Solution xiii

2.2.1 Global Planner xiii

2.2.2 Local Planner xiv

3 Test **xv**

3.1 Test Path xv

3.2 Validation: Automatic Initialization Policy xviii

3.3 Validation: Planners configuration xx

3.3.1 Simulation Tests xx

3.3.2 Real-world Tests xxix

3.4 Conclusions & Future Works xxxiii

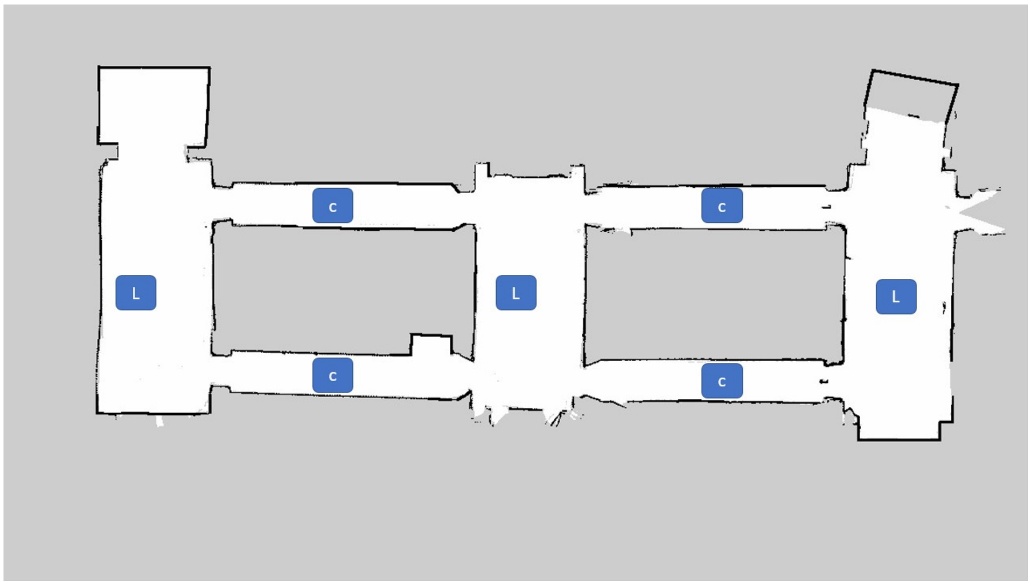


Figure 1: Map of the environment

1 Introduction

In this section, I am going to present a brief introduction about the faced problem, the following topics will be touched **(a)** general problem description (Sec. 1.1), **(b)** general description of possible solutions (Sec. 1.2).

1.1 Problem Description

The project consists of designing and implementing the control software for a mobile robot in order to endow it with the ability to move **autonomously** in a given, a **priori known**, environment.
Regarding the environment in which the robot will move, it is a portion of the first floor, of the E1 building, of the University of Salerno, shown in *Figure 1*.
As we can see, the map is characterized by two main components, that are:

- A set of 4 corridors, labeled with **C** in the map,
- A set of 3 lobby, labeled with **L** in the map.

As I will explain in the next sections, although the environment may seem very easy to explore and navigate, the fact that the elements that belong to the same component are quite similar each others, especially the different corridors, leads to a **challenging localization problem**.

Regarding the used hardware platform, it is represented by the *Turleblot3* robot[1], that is an open-source mobile robotic platform, controllable through ROS (Robotic Operating System) [Sta20].
About the **navigation** part, the problem described in *Definition* 1.1 has to be solved.

Definition 1.1 (Navigation Problem)**.** Given a *path*, represented through a set of *waypoints*, the robot has to reach the final destination as fast as possible, by passing trough all the intermediate waypoints.

A waypoint is defines as a tuple (x^{map}, y^{map}). The first element corresponds to the x-coordinate with respect to the map frame, and the second element corresponds to the y-coordinate with respect to the map frame. As we can see, the waypoint lacks of a third component, that is the orientation of the waypoint itself, that defines the attitude of the robot once the desired position is reached, the way this component has been retrieved in the project will be discussed in *Sec* 3.1.

1.2 Possible Solutions

In order to solve the given problem, that is essentially a *Navigation task*, I have to handle with two strictly related sub-problems:

1. **Localization** task, i.e. the robot must be able to localize itself within the map in the most efficient and effective way as possible, in order to generate the most suitable path that links two consecutive waypoints.

2. **Planning** task, i.e. given a source waypoint and a target waypoiny, the goal is to generate the path that allows the robot to go from the source to the target, based on the needs, the path can be optimized with respect to different aspects (e.g. minimum time, minimum traveled distance etc....). Moreover this task can be further divided into two sub-tasks:

   (a) **Global-path** planning, solving this task means generating the *global path*, i.e. the path that goes from the source to the destination, by taking into account the optimization goals, and the static obstacles,

[1]https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

(b) **Local-path** planning, solving this task means implementing the reactive components of the navigation module, i.e. generating a sequence of velocity and acceleration commands to send to the controllers of the wheels, with the aim to follow as strictly as possible the global path, while avoiding collision with dynamic and static objects.

Localization: In order to solve the localization problem, there are three main approaches:

- **Markov based localization**, this approach is theoretically the optimal one, because it does not make any assumption about the underling belief state distribution and about the initial state. However, it is computationally expensive, since, given a discrete representation of the map (e.g. occupancy grid, as in our case), Markov assigns at each cell a probability, and at each iteration the algorithm has to update the probability assigned to each cell.

- **Monte Carlo based localization** (MCL), this approach tries to be a trade-off between the Markov and the Kalman approach. Indeed, MCL is a *multi-hypothesis* approach, like Markov, however it does not keep a probability for each cell of the map, but the global belief is represented by a **dynamic set of independent particles**, each of ones representing a single state hypothesis.

- **Kalman based localization**, while it has been one of the most studied and used approach for sensor fusion, since it solves all the computational problems of Markov, this approach is the one that makes the most **strictly assumptions**. Indeed, it is a *single-state hypothesis*, distributed according to a Gaussian distribution, as well as the measurement model, and the motion and measurement errors. Moreover, the initial state is supposed to be known, with an high level of confidence.

Theoretically, all the three approaches are suitable for the given problem. However, I decided to further explore only the Monte-Carlo based localization approach, and the Kalman based localization approach. Since, the latter is the most efficient algorithm, and, from a theoretical point-of-view, the strictly assumptions are not a problem, especially about the initial position, since the robot, during the test is placed in **proximity** of the first waypoint. Instead, the former, is the most used approach in actual applications, since it is more general, and well suited for **occupancy-grid** maps. The Localization module design and implementation will be further explained in *Sec. 2*.

2 Proposed Solution

In this section, I am going to report the proposed solution, starting from the *localization problem* (*Sec.* 2.1), till the *planning problem*(*Sec.* 2.2).

2.1 Localization: Proposed Solution

Concerning the *localization problem*, the main challenge is the initial localization, i.e. understanding where the robot is is at the start-up. As already said, to solve this task two possible solutions exists **(a) Extended Kalman Filter** (EKF) (Sec. 2.1.1), **(b) Monte–Carlo Localization** (MCL) (Sec. 2.1.2). During my project I started with the implementation of the EKF. However, as I will explain in Sec. 2.1.1, the resulting Filter was quite unstable. This instability will be qualitatively proved, and some considerations about the reason of this instability will be given. So, after discarding the EKF solution, I solved the localization task with an MCL approach; in particular, I used the ROS implemented *AMCL*[2], the working mechanism and all the hyper-parameters will be explained in *Sec 2.1.2*

2.1.1 Extended Kalman Filter

As already said, at the beginning the robot is placed in proximity of the first waypoint, so the first idea is to follow the Kalman assumption and try to implement the ***Extended Kalman Filter***. The implementation follows the theoretical background reported in [SNS11]. In the cited book, the EKF is used in the following assumptions:

- The used robot is a *differential-robot* as in our case,
- The *robot belief* (state) is composed of the following components $\begin{pmatrix} x \\ y \\ \vartheta \end{pmatrix}$, i.e. the planar position components and the planar orientation (the rotation about the axis orthogonal to the plane),

[2]http://wiki.ros.org/amcl

- The map is assumed to be continue, especially a *line-based map* is used.

Based on the given assumptions, the main problem to solve is related to how obtain the lines representing the map, starting from the given occupancy-grid.

Line Extraction In order to extract the lines needed for the measurement and update step of the EKF, *3 theoretically possible solutions* have been identified:

1. Collect the laser scan over the entire space of interest, create the corresponding Point Cloud (PC) of the given measurements, and run a regression algorithm such as the *Least-Square Estimation Algorithm* to obtain the desired lines,
2. Given the current occupancy-grid map, iterate over the given matrix and identify the starting and ending point of a possible line, based on the cell value,
3. Given the current occupancy-grid map image, run classic *Image Processing Algorithms* such as *Canny Edge Detection Algorithm* and *Hough Transform Algorithm*.

Actually, the first solution is the optimal one, but the estimated time required to collect the laser scan, convert into the corresponding (PC), and implement the regression algorithm was too high for the available resources. The second and the third solution are quite comparable each others. I decided to follow the third proposed solution, since it is based on well known algorithms.
At the end, the followed procedure was:

1. Starting from the original image, apply a *Gaussian Blur Filter*, to reduce the high frequency components, and improving the output of the edge detection algorithm,
2. Starting from the filtered image, run the *Canny Edge Detection Algorithm*, to extract the edges from the image,
3. Starting from the output of step-2, run the *Hough Transform Algorithm* to obtain the lines. In the actual implementation the *Probabilistic Hough Transform* variant was used,
4. The output of step-3 is a set $P = \{((x_{start}, y_{start}), (x_{end}, y_{end}))\}$, that is a set of starting and ending point of the extracted line. For each couple of points in the set

$\{((x_{start}, y_{start})_i, (x_{end}, y_{end}))_i\}$, the general line equation $ax + by + c = 0$ was obtained, and starting from the coefficients a, b, c the corresponding polar representation (r, θ) was obtained.

The qualitative results of the steps above are reported in *Figure* 2. I obtained a total of 306 lines, due to this high amount of lines, I decided to implement a filtering procedure, that removes those lines similar each other.
The similarity is computed by checking whether the radius and angle differences of two lines are under a given threshold or not. For more detail check the file *line_extractor.py*.

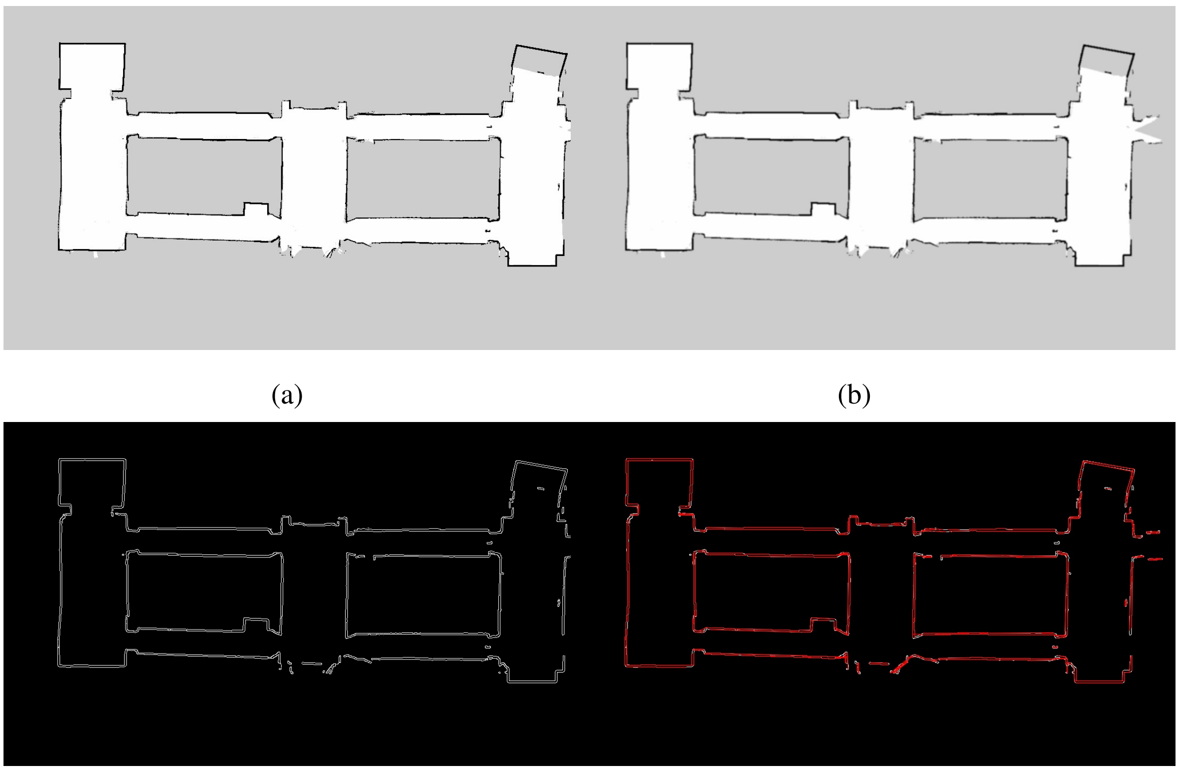


Figure 2: Line extraction steps: (a) Original Image. (b) Guassian filtered image. (c) Output of edge detection. (d) Output of Hough algorithm.

EKF Implementation Once a continue map representation is obtained, the Extended Kalman Filter can be designed and implemented. The theoretical background used for the implementation is the one reported in [SNS11]. The *EKF* is essentially composed of the following four steps:

- *Robot state prediction.* This step aims to predict the new robot state based only on the motion model, i.e. given in input the command and the previous state compute the new robot pose. The used motion model is the *non-linear model* reported in [SNS11]. For more detail check the function *KalmanFilter.py:prediction*

- *Observation.* This step aims to compute the *measured lines*, z_i, defined with respect to the local robot frame, starting from the raw-sensor values. In order to obtain such lines, a *line extraction algorithm* must be executed. For this step, I decided to rely on the publicly available ROS node *laser_line_extraction*[3] which essentially implements a line-extraction algorithm based on two steps **(i)** *Merge-and-Split*, in order to create the sets of points that belong to the same line, **(ii)** *Line-Fitting*, in order to obtain the line parameters, given the points obtained in the previous step. The line regression is based on the algorithm proposed by [PRB03], which uses a *Maximum-Likelihood* approach to estimate the line parameters $L(R, \alpha, S)$. Where $(\cdot)$ (R, α) is the vector to the normal of the infinite line, $(\cdot)$ S is the center of the line segment tangential to the infinite line.

- *Measurement Prediction.* This step aims to compute the *predicted observations*, $\widehat{z}_j$, by transforming the features map defined in the *World Frame* into the corresponding features with respect to the local *Robot Frame*. To perform such transformation, for each map line (r_j, θ_j), the transformation proposed in [SNS11] was used. For more detail check *KalmanFilter.py:_measurement_prediction* function.

- *Matching.* This is the final step, in which the sensor fusion is actual performed, also in this case the steps proposed in [SNS11] were implemented. In detail, to select a valid match between the predicted line and the measured one the *Mahalanobis distance* was used, and the *validation gait* was set to 3.219, which corresponds to the 80-th percentile of a *chi-square distribution* with 2 degree of freedom. For more detail check *KalmanFilter.py:update* funcion.

EKF test: After the implementation of the proposed Extended Kalman Filter, a set of tests have been performed in order to assess the performance of the implemented code. The evaluations were **purely qualitative**, and performed **only in simulation**. During the tests, the EKF was initialized with a known state, and either the *Path_1* (*Figure* 5a)

[3]https://github.com/kam3k/laser_line_extraction

was executed, or the robot was controlled manually through the keyboard. At the end of these tests, I concluded that, although the filter was able to update the robot state, with a consequent drop of the covariance, I observed that when the robot went thought particular areas of the map such as the entrance of a corridor, or those areas with a pour map quality, the updated state differed from the actual state. This behaviour is depicted in *Figure* 3. To attenuate this problem, I gave more importance to the odometry by reducing the validation gate, and the k_r - k_l constants of the odometry covariance matrix. Obviously, this led to a more stable behaviour, but at the same time, very few state updates were performed with respect to the motion frequency.

The reason for this unstable behaviour has not been completely found, although much effort has been put into debugging all the steps and fine-tuning the EKF's hyper-parameters. Anyway, some considerations may be proposed:

- The followed line extraction procedure is not the ideal one, since the lines are obtained from the occupancy grid map, so we should consider intrinsic error due to the map resolution ($5cm$),
- The map itself is quite challenging for a line-based representation, since in some areas of the map, such as the beginning of a corridor, there are very little vertical lines that are not detected by the line extractor node, because either there are no points or they are too few.

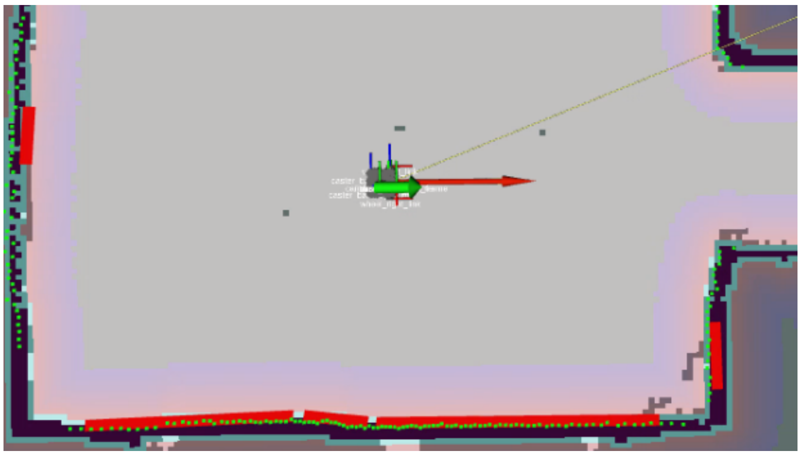
2.1.2 Adaptive Monte-Carlo Localization

After the implementation of the EKF, and observing its instability, I decided to solve the *Localization Problem* by using the Monte-Carlo approach. Specifically, I used the ROS module *AMCL*, with the following configuration:

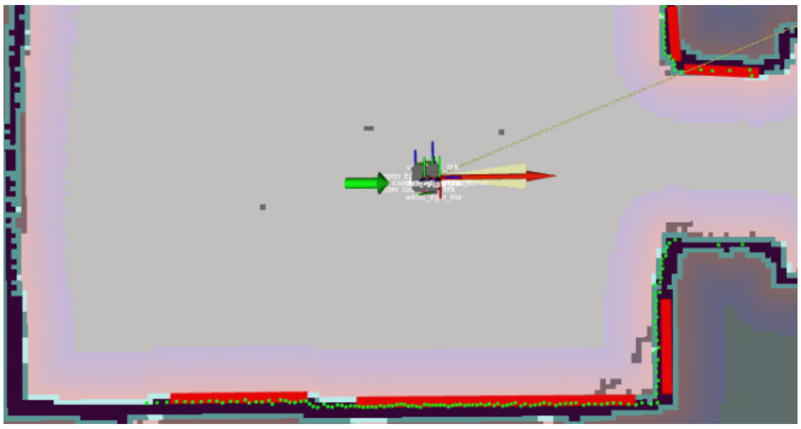
- **Probabilistic Measurement model**: *likelihood_field_range_finder_model*,
- **Probabilistic Odometry model**: *sample_motion_model_odometry*.

Both the models are explained in detail in [TBD05].

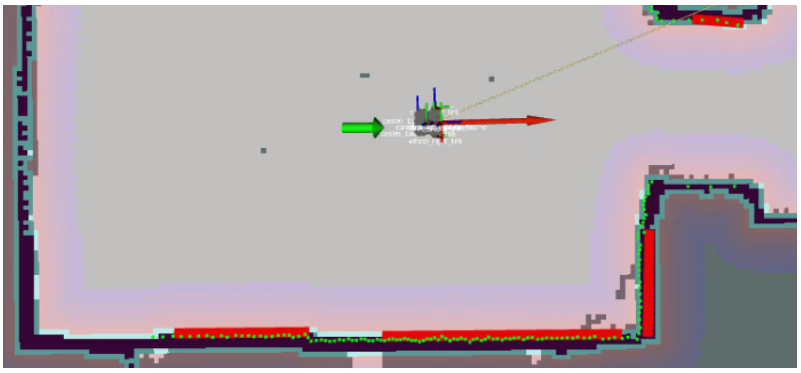
Probabilistic Measurement Model: Generally speaking, a *Probabilistic Measurement Model* has the aim to compute the probability $p(z_t^k|x_t, m)$, i.e. the conditional probability



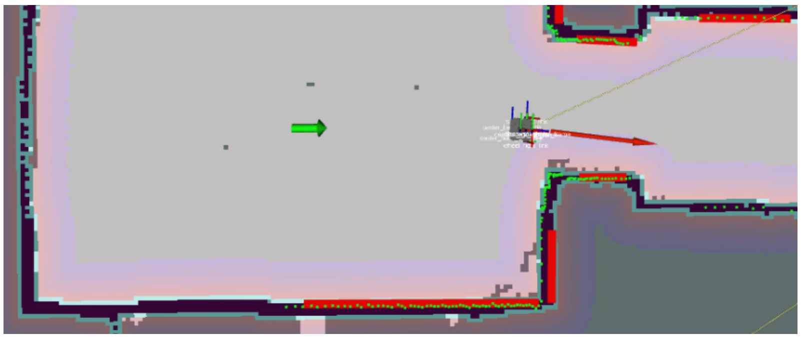
(a) Initial Condition



(b) Odometry Update: The red arrow is aligned with the robot, and the variance is increased.



(c) Correct Kalman Update: The red arrow is aligned with the robot, and the variance is decreased.



(d) Divergence: The red arrow is not aligned with the robot.

Figure 3: Sequence of *Extended Kalman Filter* behaviour. The red arrow represents the predicted pose.

of getting the measure z_t^k, given the current predicted state x_t and the map m. For the selected model, this probability is a mixture of **3 different distributions** that take into account the following type of errors:

- **Local measurement error**: This model tries to take into account the fact that even if the sensor measures the range to the nearest object the returned value is subject to error. The model is defined as follow: $p_{hit}(z_t^k|x_t,m) = \varepsilon_{\sigma_{hit}^2}\left(\mathbf{dist}^2\right)$, where ε is a zero-centered Gaussian distribution with variance σ_{hit}^2, and $\mathbf{dist}^2$ is the Euclidean-distance between the projected measure $\left(x_{z_t^k}, y_{z_t^k}\right)$ and the nearest object in the map (e.g. the nearest occupied cell).

- **Failures**: This model takes into account the error generated when an obstacle is missing, and it is formulated as follow:
$$p_{max}(z_t^k|x_t,m) = I(z = z_{max}) = \begin{cases} 1 & if\ z = z_{max} \\ 0 & otherwise \end{cases}$$

- **Random measurements**: This model takes into account the unexplained measurements produced by the noisy sensor, and it is formulated as follow:
$$p_{rand}(z_t^k|x_t,m) = \begin{cases} \frac{1}{z_{max}} & if\ 0 \le z_t^k \le z_{max} \\ 0 & otherwise \end{cases}$$

At the end, the final probability is given by:

$$p(z_t^k|x_t,m) = \begin{pmatrix} z_{hit} \\ z_{max} \\ z_{rand} \end{pmatrix}^T * \begin{pmatrix} p_{hit}(z_t^k|x_t,m) \\ p_{max}(z_t^k|x_t,m) \\ p_{rand}(z_t^k|x_t,m) \end{pmatrix}$$

From this discussion, 4 hyper-parameters can be identified, that are z_{hit}, z_{max}, z_{rand}, σ_{hit}^2. These parameters will be fine-tuned during the tests.

Probabilistic Odometry Model: Generally speaking, the aim of a Probabilistic Motion Model is to compute $p(x_t|x_{t-1}, u_t)$, i.e the probability that starting from the state x_{t-1} and applying the command u_t the system goes to the state x_t.
In *sample_motion_model_odometry*, rather than compute the conditional probability, a "randomized" robot state x_t is directly computed. This approach is particularly useful when a Particle Filtering approach is used, since the localization module does not need to solve a closed problem for each particle to compute $p(x_t|x_{t-1}, u_t)$, but instead it can modify the predicted robot state by adding a **random error term**. The algorithm is reported in *Figure* 4, and it can be roughly divided into three steps:

1. The measured movement, i.e. the difference between the previous measured state $(\bar{x}, \bar{y}, \bar{\theta})$ and the current measured state $(\bar{x}', \bar{y}', \bar{\theta}')$, is expressed in terms of two rotational components e one translational component, LINE 2-4.

2. The computed motion components are modified by adding random error, LINE 5-7.

3. The predicted final state is obtained by adding to the previous state (x, y, θ) the randomized measured motion components, LINE 8-10.

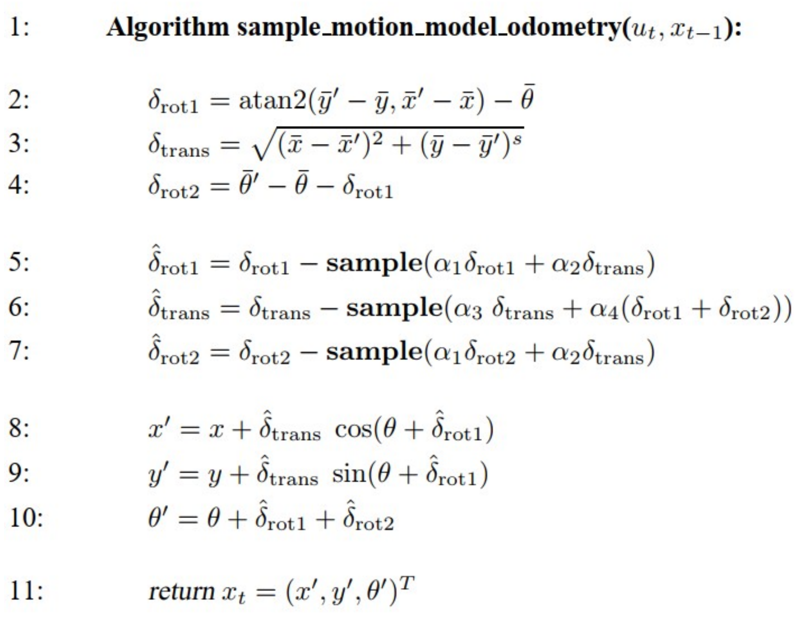
The parameters α_1, α_2, α_3, α_4 are hyper-parameters that have to be fine-tuned during the tests.

2.1.3 Bring-up: Automatic Localization Procedure

After the discussion in *Sec.* 2.1.1 and *Sec.* 2.1.2, I solved the localization problem with the AMCL module, however one question must be answered *"What happens at the start-up of the robot?"*.
From the problem definition I known that the robot is placed in the vicinity of the source waypoint, however, the localization node needs to be initialized. Generally speaking, there are at least three possible solution to this problem:

1. **GUI Initialization**: Initialize the AMCL pose by using the graphical user interface provided by *Rviz*,

Figure 4: sample_motion_model algorithm



```
1:    Algorithm sample_motion_model_odometry($u_t, x_{t-1}$):

2:        $\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
3:        $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^s}$
4:        $\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

5:        $\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta_{\text{rot1}} + \alpha_2 \delta_{\text{trans}})$
6:        $\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3\ \delta_{\text{trans}} + \alpha_4(\delta_{\text{rot1}} + \delta_{\text{rot2}}))$
7:        $\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta_{\text{rot2}} + \alpha_2 \delta_{\text{trans}})$

8:        $x' = x + \hat{\delta}_{\text{trans}}\ \cos(\theta + \hat{\delta}_{\text{rot1}})$
9:        $y' = y + \hat{\delta}_{\text{trans}}\ \sin(\theta + \hat{\delta}_{\text{rot1}})$
10:       $\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

11:       return $x_t = (x', y', \theta')^T$
```

2. **Manual Initialization**: Initialize the AMCL pose by using the *turtlebot3_teleop* node[4], which allows the user to teleoperate the robot through the keyboard. With this solution I could move the robot until the estimated pose reaches the desired confidence level,

3. **Automatic Initialization Policy**: This solution is based on the idea that the robot should follow a policy that allow it to localize itself within the map, i.e. the estimated pose reaches a certain confidence level.

The most challenging solution is the third one, and I decided to implement such policy to implement the desired behavior. This policy aims to move the robot in the direction with the **highest probability of gaining information** about the environment while keeping the robot itself safe with respect to the obstacles.
To gain information about the environment, the robot should **explore the unknown space**, that can be identified by the laser-scan values, indeed, if the robot moves in the direction of the **max-sensor reading** this means that it potentially explores unknown space. This condition is met when the max-sensor reading is equal to *inf* (exploring), but, even if it is not, the robot moves towards the direction of the highest distance from an obstacle (safety). After some trial-and-error steps, I ended up with the policy described by the

[4]http://wiki.ros.org/turtlebot3_teleop

following steps:

1. Get the current laser scan measurement $laser\_scan\_values$,
2. Get the index of the maximum measurement $i = \text{argmax}\, laser\_scan\_values$,
3. Align the robot with the direction of the maximum measurement,
4. Move the robot for a space equal to $linear\_motion\_space = laser\_scan\_values[i] - safety\_value$
5. Perform one $360deg$ rotation,

The procedure above is repeated until a certain confidence is reached in the estimated pose. In detail, I check $\sigma_x^2, \sigma_y^2, \sigma_{yaw}^2$ obtained from the covariance matrix of the estimated pose. For more detail check *main.py:automatic_localization_procedure* function.

2.2 Planning: Proposed Solution

In this section, the *Planning Module* is under discussion. In particular, the two planning modules, that are: **(a)** Global Planner (GP) (Sec. 2.2.1), **(b)** Local Planner (LP) (Sec. 2.2.2), will be discussed. Regarding both the GP and the LP, I used the ROS packages already integrated in the **ROS navigation-stack**.

2.2.1 Global Planner

The aim of the Global planner is to generate the path that allow the robot to go from a starting position to a goal position, while avoiding obstacles and optimizing a given cost function. The problem of finding the optimal global path can be formulated in terms of finding the shortest path in a given graph, which is a well-studied problem and for which a vast amount of algorithms exist. Indeed, the ROS packages *global_planner* and *navfn* solve the global planning problem by using the optimal *Dijkstra* algorithm or the heuristic-based $A*$ algorithm.
Both, Dijkstra and A* use the same cost function that is $cost = COST\_NEUTRAL + COST\_FACTOR * costmap\_cost\_value$, also in this case we have essentially two hyper-parameters $COST\_NEUTRAL$ and $COST\_FACTOR$.

2.2.2 Local Planner

As concerns the Local Planner, its goal is two-fold, first of all it has to generate the velocity commands to send to the motor controllers that allow the robot to move and follow the planned global path, while realizing the reactive component, i.e. avoiding dynamic obstacles. In this case the ROS navigation-stack provides different packages, among which, I selected: • *dwa_local_planner*[5], • *teb_local_planner*[6] Few words are going to be spent for each of one.

DWA Local Planner: The *Dynamic Window Approach* for local planning, proposed in [FBT97] is the basic approach for solving the problem, it follows a *sample-and-forward paradigm*, essentially this algorithm performs the following steps:

- The control space is discretized, and within this discrete space a set of linear and angular velocities are sampled $(dx, dy, d\theta)$,
- The sampled control inputs are applied for a short period of time,
- The generated trajectories are ranked based on a cost function that takes into account the proximity to the obstacles, the proximity to the global path and global waypoint, moreover the trajectories that collide with the obstacles are discarded,
- The trajectory with the highest score is selected, and the corresponding command are sent to the robot.

TEB Local Planner: The *Time Elastic Band* method has been proposed for the first time in [Rös+12], it represents an improvement of the *Elastic Band* method, and, more in general, of all the local planner methods. Briefly, this method considers the kinematic and dynamic motion constraints, and model the local path as a sequence of n intermediate robot poses $x_i = (x_i, y_i, \theta_i)^T$. Moreover, to generate the local plan, it solves a multi-objective optimization problem, defined as $B^* = \text{argmin}_B \sum_k \gamma_k f_k(B)$, where f_k is the k^{th} objective function, which can belong to two categories:

- *Penalty functions*, for velocity and acceleration constraints,

[5]http://wiki.ros.org/dwa_local_planner

[6]http://wiki.ros.org/teb_local_planner

- *Objective functions*, for desired characteristics such as shortest or fastest path, clearance from obstacles, and so on.

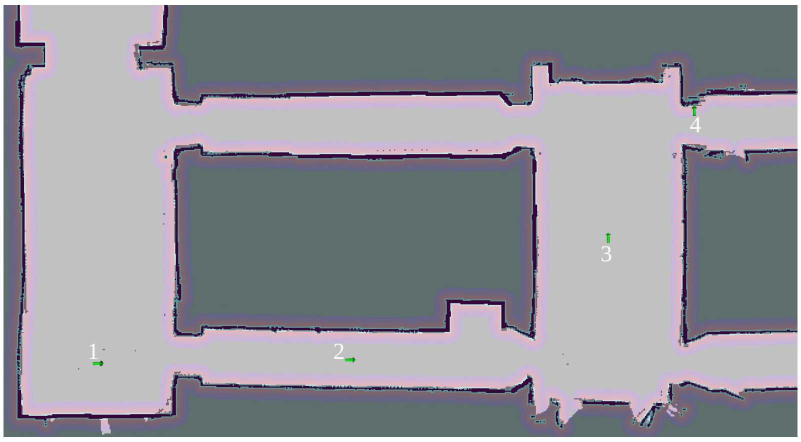
Before starting with the test, I performed a preliminary study by reading articles that propose a comparison between different global/local planner configurations [Pit+18; Fil+20; NW20]. As expected, there is not a configuration that is the best one in all the possible cases, and the decision between the DWA and TEB approach must be taken case by case.

3 Test

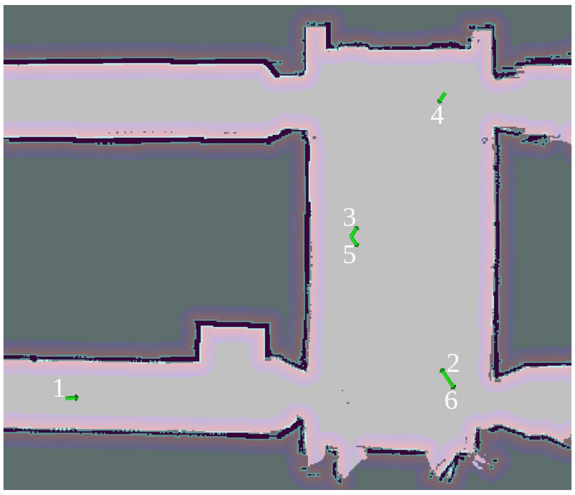
In this section I am going to present the test performed to validate and fine-tune: **(a)** The automatic localization procedure (*Sec.* 3.2), **(b)** The global/local planner configuration (*Sec.* 3.3) I will start by introducing the paths used for testing *Sec.* 3.1. Keep in mind that the tests have been designed and performed according to the classic robotic software deployment procedure, i.e. the tests have been performed first in **simulation**, then the most interesting ones have been performed also on the **real robotic platform**.

3.1 Test Path

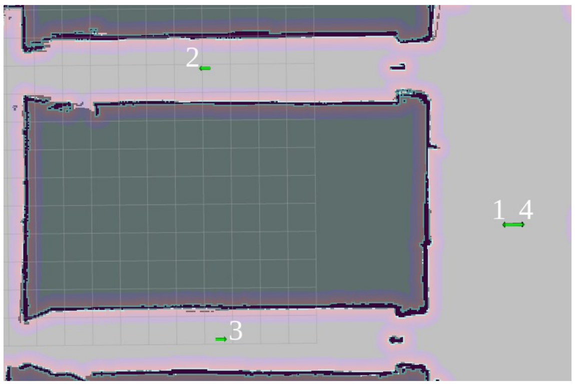
In this section, the paths used for the tests are going to be discussed. I designed three paths for testing the proposed localization/planning configuration *Figure* 5, and other two paths to test the waypoint orientation definition *Figure* 6.
The *Path_1* (Figure 5a) is the simplest one, essentially it has been proposed as a first benchmark for the proposed configuration. The *Path_2* (Figure 5b) is a little more complex, and it has been proposed for two reasons, **(i)** Test the automatic localization procedure when the robot is placed in a corridor, **(ii)** Test the proposed configuration in a setting where the path is composed of changes in directions. The *Path_3* (Figure 5c) has been proposed in order to: **(i)** Test a longer path, **(ii)** Verify the behaviour of the *localization module* during such long path, **(iii)** Verify the behaviour of the *local planner* with respect to the presence of obstacles.
By looking at the figures, an arrow represents a waypoint, and it is characterized by its positional and rotational components. The last component is not present in the waypoint



(a) *Path_1*

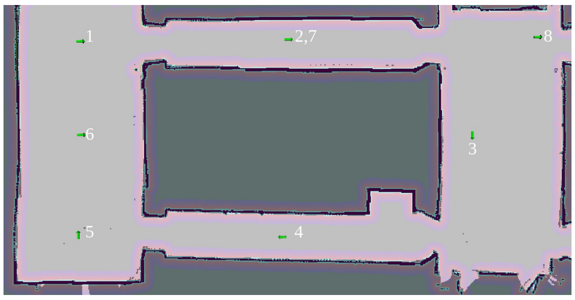


(b) *Path_2*

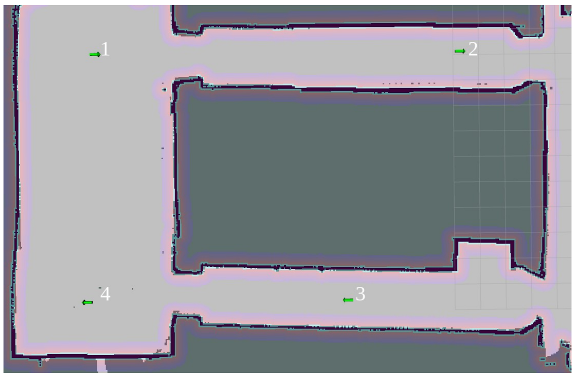


(c) *Path_3*

Figure 5: Paths used for testing the proposed configuration.



(a) *Path_4*



(b) *Path_5*

Figure 6: Paths used for testing the waypoint orientation.

definition given in *Sec.* 1.1, so *how has it been obtained?*. Also in this case different solution exists:

- For each waypoint, set the orientation to 0. This is the simplest solution, however, the robot will align the $x_{base\_footprint}$ axis with the x_{map} axis, once the waypoint is reached. This is not a problem for *Path_1*, where the motion direction is always toward the positive x_{map} direction. It is a very important problem for *Path_2* and *Path_3*, especially for the last one, where the direction of the motion is along the negative x_{map} direction, from wp_1 and wp_2.

- For each waypoint, set the orientation with respect to the next waypoint ($\theta_{wp_i} = arctan(\frac{|wp_{i+1} - wp_i|_{2,y}}{|wp_{i+1} - wp_i|_{2,x}})$. This is a first improvement with respect to the first solution, but this approach leads to very undesirable results for paths like the *Path_3*. In this case the orientation of wp_2 will bring the robot to point toward the wall, and, consequently there will be a wasting of time due to the realignment of the robot with the global path.

- For each waypoint, set the orientation in such a way to minimize the misalignment between the waypoint orientation and the global path. This can be seen as a chicken-and-egg problem, since to obtain the global path I need a complete defined set of waypoints (i.e. position and orientation), while to compute the error between the path and waypoint orientation I need the global path.

As I will show in *Sec.* 3, in the very beginning I used the alignment policy based on the second bullet-point, then I moved on the final proposed solution. The proposed solution is essentially based on a combination of the second and third bullet-point. In order to break the dependency between the global-path and the waypoints, I performed a **pre-processing** of the given **set of waypoints**. During this pre-processing the following conditions are checked:

- Whether there is an obstacle between the waypoints or not,
- Whether two consecutive waypoints are placed in parallel corridors or not.

Based on the following conditions, I determine the waypoint orientation based on the following reasoning:

- If there is an obstacle, and there is a corridor between the points, set the orientation equal to the same orientation of the previous waypoint, in order to follow the possible path (e.g. *Path_5* from 2 to 3, *Figure* 6b),
- If there is an obstacle, but there is not a corridor between the points, set the orientation equal to one of the following values $\{0, \frac{\pi}{2}, -\frac{\pi}{2}, \pi\}$, based on the fact that the relative orientation is within a given range, defined for each value (e.g. *Path_4* from 3 to 4, *Figure* 6a)),
- If there are not obstacles, then set the orientation equal to $\theta_{wp_i} = arctan(\frac{|wp_{i+1} - wp_i|_{2,y}}{|wp_{i+1} - wp_i|_{2,x}})$ (e.g. *Path_2 Figure* 5b.

For more detail check the *utils.py:compute_wp_orientation* function.

3.2 Validation: Automatic Initialization Policy

In this section I am going to discuss the tests and the fine-tuning performed for the localization module, and, especially for the Automatic Initialization Policy. In order to achieve a valid initialization I identified different hyper-parameters that could affect the localization quality.

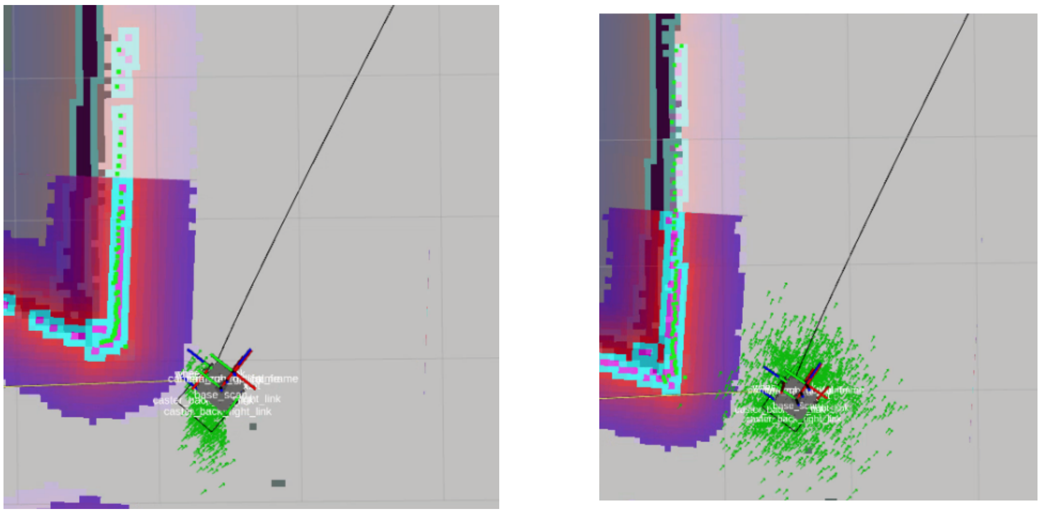
More in detail, during the tests I observed a quite large misalignment between the map and the partial point-cloud produced by the laser scan, as depicted *Figure* 7a. This behaviour might be an effect of a wrong localization, so I decided to modify the parameters of the AMCL module in order to achieve better performance.

While at the very beginning, I thought that this problem could be solved by increasing the number of the particles and their variance, in the end, after studying the *Probabilistic Measurement Model*, the *Probabilistic Odometry Model* (*Sec.* 2.1.2) and the corresponding configuration parameters (*amcl.yaml*), I observed how the behavior of the localization module changes a lot by modifying the following parameters:

- **laser_likelihood_max_dist**: This parameter defines the maximum distance between the measured point and the nearest obstacle point on the map to take into account for computing the likelihood $p_{hit}(z_t^k|x_t,m)$. By reducing this distance, we reduce the possibility to take into account possible outliers. Indeed, from the default value of 2.0 *m*, at the end I used 0.5 *m*.

- **odom_alpha1**, **odom_alpha2**, **odom_alpha3** and **odom_alpha4**: These parameters define the *"amount of error"* to consider during the pose sampling (*Figure* 4). Increasing these value lead to a **more noisy odometry model**. The occurrence of this statement could seem undesirable, however it has also the following interpretation, by increasing the error term in the algorithm in *Figure* 4, the sampled poses are more scattered, this means that during the motion of the robot I reduce the *loss of diversity*. Indeed, as It can be seen from *Figure* 7b the particles are much more spread around the robot, with respect to *Figure* 7a, that is the AMCL with the default parameters. Moreover, the discrepancy between the wall of the map and the points obtained from the laser scan is lower with the custom parameters, than with the default parameters.

There is a formal procedure to determine all the parameters of both the measurement model and odometry model, explained in [TBD05], however for reasons of time and resources I did a simple trial-and-error approach. In conclusion, from now on, the **AMCL with custom parameter** () must be considered as the proposed solution for the localization problem.



(a) AMCL with default parameters

(b) AMCL with custom parameters

Figure 7: AMCL parameters influence on localization. (a) Default Parameters, (b) Custom Parameters

3.3 Validation: Planners configuration

In this section I am going to describe the tests performed to validate the global/local planner configuration. As already said I divided the tests into two groups **(a)** Simulation Tests (Sec. 3.3.1), **(b)** Real-world Tests (Sec. 3.3.2). The first group aims to identify the most interesting set of parameters, while the second group aims essentially to validate the tests performed in simulation, and, eventually, highlight the discrepancy between the simulated world and the real one.

3.3.1 Simulation Tests

I am going to start the discussion about the tests, starting from the simulated one. As already said the paths used are the ones depicted in *Figure* 5. For each test I measured the time spent to complete a single path together with an average time over the three paths. Also in this case, the tests are divided into two groups **(a)** *Out of the box tests*, **(b)** *Real-world Tests*.
The former tests aim to evaluate the configurations as they are, with the default parameters in order to set a **baseline**, the latter tests aim to fine-tuning the parameters of the most interesting configuration obtained from above.

Out of the box tests Table 1 reports the first tests performed in simulation, with default parameters. Only the third test has a different *Obstacle distance* needed to avoid the robot stuck in the entrance of the corridor where there is an obstacle in the middle. As it can be noted the *teb* local planner has better performance, this meanly for two reasons: **(i)** *DWA* tends to follow the global path as strictly as possible, while *TEB* is able to optimize the local path also with respect to the path length. Consequently, when the robot performs a turn, with *TEB* it is near the wall, keeping the minimum obstacle distance, instead, with *DWA* the robot performs more larger turns due to the inflation radius that bring the global path far from the wall. **(ii)** With respect to *DWA*, *TEB* explicitly solves an optimization problem, and it is not based on a sampling approach. This means that, the path built with *TEB* is characterized by velocities and accelerations that are near to the limit, if it is necessary for the optimization goal.
After these preliminary tests, I modified the waypoint orientation by using the policy explained in *Sec.* 3.1, and I obtained the results in *Table* 2. As it can be observed, there is not a particular gain in performance for TEB, since it is able to produce very fast in-place rotations. At the same time, we can see a remarkable improvement when DWA is used, especially on *Path_3*, because now the robot do not need to perform in-place rotation for the alignment with the waypoint. However, TEB continues to perform better than DWA.
At the end of these tests **I selected TEB as local planner**, and I observed that, although the performance in terms of time were better than DWA, by looking at the robot behaviour it swayed as it moved along the path. Indeed, by looking at the plots in *Figure* 8 related to *Path_1* that is the simplest one, it can be noted that while the linear velocity is always at the upper limit, the angular velocity has high and relevant oscillations. This phenomenon could be caused by two aspects: **(a)** The Global Planner and the smoothness of the generated path, **(b)** The Acceleration Limit of the Local Planner.
So I decided to perform tests by fixing the local planner and changing the global planner, and by changing the acceleration limit of the local planner.

Regarding the Global Planner, as already explained in *Sec.* 2.2.1 there are essentially two possible solutions: **(i)** A*[7], an heuristic based algorithm that solves the problem of finding the optimal path given a graph, **(ii)** Dijkstra[8], an optimal algorithm that solves the same problem as A* but without an heuristic approach. From the results, reported in

[7]http://wiki.ros.org/global_planner
[8]http://wiki.ros.org/navfn?distro=noetic

Table 3, it can be noted how changing the parameters of Dijkstra has not a relevant impact on the performance. However, changing from Dijkstra to A* has a tragic consequence on those paths that involves a turn close to an obstacle, which can be an obstacle or the wall, (e.g. *Path_3*. Indeed, the heuristic algorithms generates paths that are very close to the obstacle. Consequently, the motion is very unstable because the local planner tends to move the path far from the obstacle but close to the global path. After these tests I decided to **keep Dijkstra as global planner**.

Regarding the acceleration limit to make the angular velocity less variable, first of all, I decided to try with different limit values. The results are reported in *Table* 4, and the velocity profile are depicted in *Figure* 9. From the obtained results two things can be noted:

- By reducing the acceleration limit, the angular velocity keeps to be highly variable, but the pick value of the fluctuation is negligible with respect to visible rotation.
- By reducing the acceleration limit, the time spent to complete a path increases, and in some cases, it becomes worse than the original DWA baseline. This is because with this small limit the local path follows strictly the global one, consequently increasing the time needed to complete it.

Interesting acceleration limits are 0.5 and 1.5, which have average time performance similar to the baseline, but with a smoother behaviour. A more in-depth analysis could have been done considering such parameters such as the weight given to the acceleration limit in the optimization problem.

In conclusion, in the next paragraph, the one dedicated to fine-tuning, the **acceleration limit equal to 0.5 and 1.5**, and **obstacle distance equal to 0.2** will be considered.

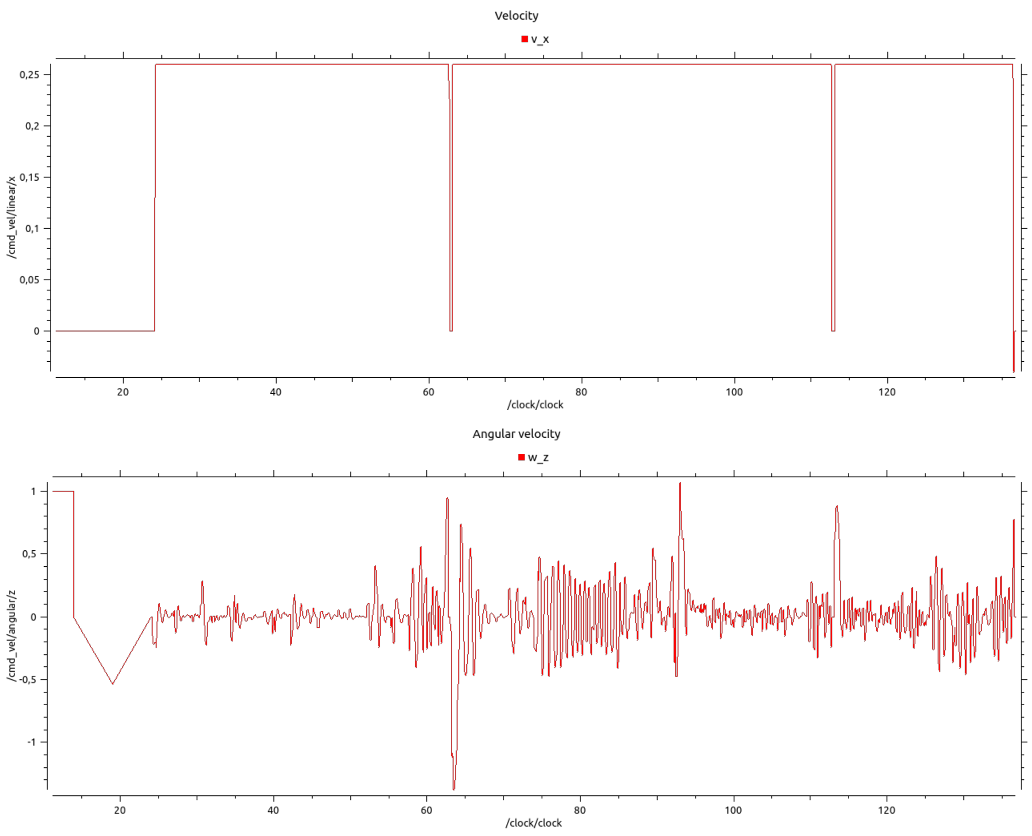
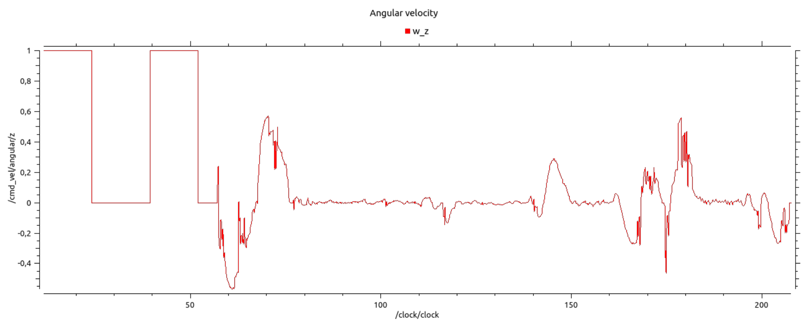


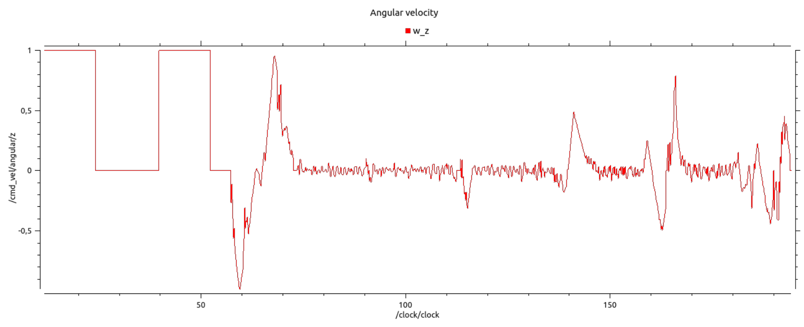
Figure 8: TEB velocity profile with default parameters

Table 1: Tests performed with default parameters, the waypoint orientation was equal to: $\theta_{wp_i} = arctan(\frac{|wp_{i+1}-wp_i|_{2,y}}{|wp_{i+1}-wp_i|_{2,x}})$.

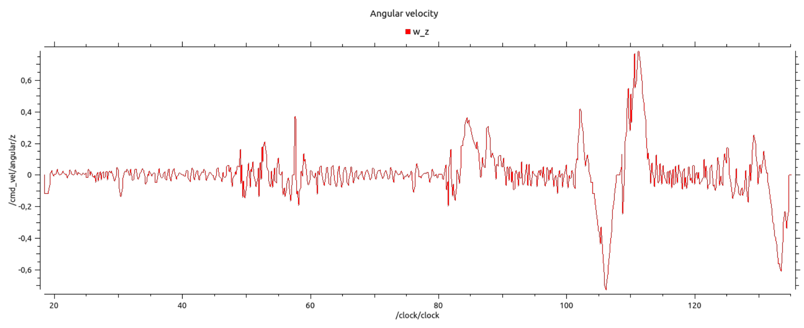
Test ID	**Global Planner**	**Local Planner**	**Time-Path 1 - SIM [m:s]**	**Time-Path 2 - SIM [m:s]**	**Time-Path 3 - SIM [m:s]**	**Average Time [m:s]**
1	navfn: Default	dwa: Default	1:57	2:37	3:41	2:45
2	navfn: Default	teb: Default	1:52	2:47	crash	Not-Defined
3	navfn: Default	teb: Default - Obstacle distance 0.2	1:51	2:18	3:13	**2:27**



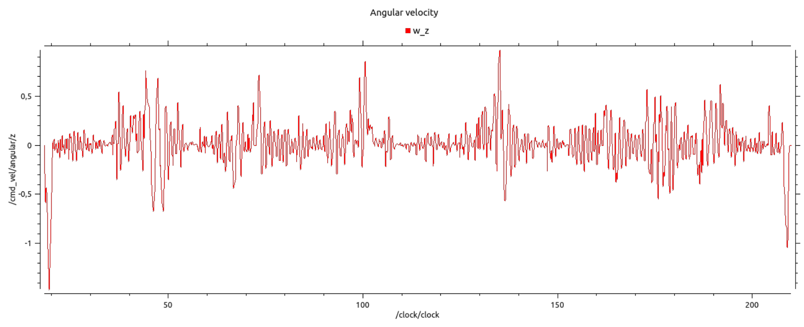
(a) Acceleration limit 0.1



(b) Acceleration limit 0.3



(c) Acceleration limit 0.5



(d) Acceleration limit 1.5

Figure 9: Angular Velocity profile with different acceleration limits

Table 2: Tests performed by using the waypoint orientation policy in *Sec.* 3.1

Test ID	Global Planner	Local Planner	Time-Path 1 - SIM [m:s]	Time-Path 2 - SIM [m:s]	Time-Path 3 - SIM [m:s]	Average Time [m:s]
4	navfn: Default	dwa: Defualt	2:01	2:29	3:22	2:38
5	navfn: Default	teb: Default - Obstacle distance: 0.2	1:52	2:19	3:11	**2:27**

Table 3: Tests performed by changing the global planner.

Test ID	Global Planner	Local Planner	Time-Path 1 - SIM [m:s]	Time-Path 2 - SIM [m:s]	Time-Path 3 - SIM [m:s]	Average Time [m:s]
6	A*: with default parameters	teb: Default - Obstacle distance: 0.2	01:55	02:20	03:49	02:41
7	A*: with parameters from [Zhe16]	teb: Default - Obstacle distance: 0.2	01:57	02:22	03:41	02:40
8	navfn: with parameters from [Zhe16]	teb: Default - Obstacle distance: 0.2	01:53	02:27	03:11	**02:30**

Table 4: Tests performed by changing acceleration limit

Test ID	**Global Planner**	**Local Planner**	**Time-Path 1 - SIM [m:s]**	**Time-Path 2 - SIM [m:s]**	**Time-Path 3 - SIM [m:s]**	**Average Time [m:s]**
9	navfn: with parameters from [Zhe16]	teb: Obstacle distance: 0.2 Acc. limit: 0.1	02:10	02:37	03:24	02:43
10	navfn: with parameters from [Zhe16]	teb: Obstacle distance: 0.2 Acc. limit: 0.3	02:00	02:35	03:46	02:47
11	navfn: with parameters from [Zhe16]	teb: Obstacle distance: 0.2 Acc. limit: 0.5	01:56	02:17	03:14	02:29
12	navfn: with parameters from [Zhe16]	teb: Obstacle distance: 0.2 Acc. limit: 1.5	01:53	02:19	03:11	**02:27**

Fine-tuning tests In this paragraph I am going to discuss the tests performed as fine-tuning, that is the tests that aim to find the best configuration of the TEB optimizer.
I have to highlight the fact that the tested parameters values have been obtained from a preliminary phase during which I used *rqt_reconfigure* node[9] to evaluate what happens when the parameters are modified.
First of all I started from an observation, the robot used to waste time on *Path_2* because it swung over the waypoint to align with it, in order to solve this problem, I followed the idea of **penalizing the backward local paths** so that the robot was forced to arrive at the waypoint already aligned. This idea brought the tests reported in *Table* 5, where I increased the *forward_drive* weight from 1 to 200. With the proposed modification the robot *3 secs* on *Test 14* on *Path_2*, and in average this test reports the best performance obtained up to now.

Now, the question is *Is it possible to obtain better performance?*. In order to try to give an answer I performed other two sets of tests, both the sets of tests have been guided by the idea to **improve how the robot arrives at the waypoint**. Indeed for all the tested paths, this represents the critical part in terms of time, since in the turns and straightforward segments, the robot has the maximum speed as possible. The two sets differs in how I tried to reach this goal, in particular:

- In the first set (*Table* 6) I forced the local planner to generate paths that follow more strictly the global path, by increasing the *wieght_viapoint* from 0 to 10, with a separation of 0.5 *m*,
- In the second set (*Table* 7) I forced the local planner to prefer shortest paths, by increasing the *shortest_path* parameter from 0 to 5.

As it can be noted neither in the first set nor in the second one I obtained some remarkable improvements. In conclusion, the best configuration obtained during the tests performed in simulation was:

1. **Global Planner**: Dijkstra implemented by *navfn* node with parameters from [Zhe16],
2. **Local Planner**: *TEB*, with *acceleration_limit*=1.5, *obstacle_distance*=0.2, *forward_drive*=200, *shortest_path*=0, and *weight_viapoint*=0.

[9]http://wiki.ros.org/teb_local_planner/Tutorials/Setup%20and%20test%20Optimization

Table 5: Tests performed to penalize the backward paths. Global Planner *navfn* with parameters from [Zhe16]

Test ID	**Local Planner**	**Time-Path 1 - SIM [m:s]**	**Time-Path 2 - SIM [m:s]**	**Time-Path 3 - SIM [m:s]**	**Average Time [m:s]**
13	teb: Acc. limit: 0.5 Forward Drive: 200	01:54	02:16	03:15	02:28
14	teb: Acc. limit: 1.5 Forward Drive: 200	01:53	02:16	03:10	**02:26**

Table 6: Tests performed by giving more weight to the global path, *with Forward Drive: 200*.

Test ID	**Local Planner**	**Time-Path 1 - SIM [m:s]**	**Time-Path 2 - SIM [m:s]**	**Time-Path 3 - SIM [m:s]**	**Average Time [m:s]**
15	teb: Acc. limit: 0.5 Global Plan Viapoint Sep: 0.5 Weight Viapoint: 10	01:54	02:14	03:16	02:28
16	teb: Acc. limit: 1.5 Global Plan Viapoint Sep: 0.5 Weight Viapoint: 10	01:54	02:18	03:13	02:28

Table 7: Tests performed by giving more weight to the shortest path, with *Forward Drive: 200*.

Test ID	**Local Planner**	**Time-Path 1 - SIM [m:s]**	**Time-Path 2 - SIM [m:s]**	**Time-Path 3 - SIM [m:s]**	**Average Time [m:s]**
17	teb: Acc. limit: 0.5 Shortest path: 5	01:56	02:19	03:15	02:30
18	teb: Acc. limit: 1.5 Shortest path: 5	01:54	02:22	03:11	02:29

3.3.2 Real-world Tests

In this section I am going to report the tests performed with the real robot. As already said these tests aim to **validate** the results obtained in simulation and eventually **highlight** the differences between the simulation-world and the real-world.

As always I started by setting a baseline, running the algorithms with the default parameters, and the best configuration obtained from simulated tests. The results are reported in *Table* 8. From the results I can make the following considerations:

- In average TEB performs better than DWA, which validates the results obtained in simulation,
- The best TEB configuration obtained in simulation performs worse than the default configuration. Although the *Test 3* setup generates smoother motion than the *Test 2* setup, where the robot swings during straight stretches. The reason of this behavior may be found in the accuracy of the real robot movements, that brings the robot to waste time during the alignment with the waypoints, producing oscillations in both the cases. However, in *Test 2*, since the robot can perform faster in place rotations, it aligns faster than in *Test 3*.

After these considerations I tried to improve the performance of the configuration in *Test 3*, since I want to avoid oscillations. Starting from the configuration in *Test 3*, I performed three tests, reported in *Table* 9, which have been designed with the aim to improve the way how the robot reaches the waypoint. As before, I modified the weight given to: **(a)** the

global path (*weight_viapoint* parameter) in *Test 4*, **(b)** the shortest path (*shortest_path* parameter) in *Test 5*. Moreover, I performed a test (*Test 6*) where I combined the configuration of *Test 4* and *Test 5*.
From *Table* 9, it can be noted how:

- By giving more weight to the global planner, we have a performance degradation, this because the robot tends to follow the global path, and essentially, in real-world DWA and TEB are equivalent,
- By giving more weight to the shortest path I observed an improvement on all the tested paths, this because The robot does not make large curves to align with the waypoint, but prefers a narrow curve followed by an in-place rotation, reaching the same performance, or even better as the default configuration.

As before, starting from the configuration in *Test 5*, the question is *Is it possible to improve the performance?*. In this case I tried to modify the following parameters:

- Give more weight to the *optimal_time* parameter (*Test 7*), in order to see what happens whether the path is composed of more rapid state transactions,
- Give more weight to the *shortest_path* parameter (*Test 8*),
- Modify the time interval considered for a state transaction (*Test 9-10*), to evaluate the influence of the sampling time in the local path computation.

As it can be seen from the results reported in *Table* 10 I did not obtain an improvement with respect to the best result in *Table* 9-*Test 5*. In conclusion the proposed solution is:

- **Global Planner**: Dijkstra implemented by *navfn* node with parameters from [Zhe16],
- **Local Planner**: *TEB*, with *acceleration_limit*=1.5, *obstacle_distance*=0.2, *forward_drive*=200, *shortest_path*=5, *weight_viapoint*=0, and *global_plan_viapoint*=-1.

Table 8: Real-World baseline

Test ID	**Global Planner**	**Local Planner**	**Time-Path 1 - RW [m:s]**	**Time-Path 2 - RW [m:s]**	**Time-Path 3 - RW [m:s]**	**Average Time [m:s]**
1	navfn: Default	dwa: Default	02:10	02:29	03:42	02:47
2	navfn: Default	teb: Default	02:03	02:27	03:30	02:40
3	navfn: with parameters from [Zhe16]	teb: Obstacle distance: 0.2 Acc. limit: 1.5 Forward Drive:200	02:05	02:32	03:35	02:44

Table 9: Tests to improve the alignment with the waypoint, with *navfn* as global planner, *acceleration_limit*=1.5 and *forward_drive*=200

Test ID	**Local Planner**	**Time-Path 1 - RW [m:s]**	**Time-Path 2 - RW [m:s]**	**Time-Path 3 - RW [m:s]**	**Average Time [m:s]**
4	teb: Global Plan Viapoint: 0.5 Weight viapoint: 10 Shortest Path: 0	02:10	02:33	03:43	02:48
5	teb: Global Plan Viapoint: -1 Weight viapoint: 0 Shortest Path: 5	02:00	02:25	03:30	**02:38**
6	teb: Global Plan Viapoint: 1 Weight viapoint: 1 Shortest Path: 5	02:01	02:27	03:30	02:39

Table 10: Tests to improve the performance of the best configuration *Test 5*, *Table* 9

Test ID	**Local Planner**	**Time-Path 1 - RW [m:s]**	**Time-Path 2 - RW [m:s]**	**Time-Path 3 - RW [m:s]**	**Average Time [m:s]**
7	teb: Shortest Path: 5 Optimal Time: 3 dt_ref: 0.3 dt_hysteresis: 0.03	02:08	02:31	03:37	02:45
8	teb: Shortest Path: 10 Optimal Time: 1 dt_ref: 0.3 dt_hysteresis: 0.03	02:07	02:33	03:38	02:46
9	teb: Shortest Path: 5 Optimal Time: 1 dt_ref: 0.5 dt_hysteresis: 0.05	02:02	02:29	03:30	02:40
10	teb: Shortest Path: 5 dt_ref: 0.1	unstable	unstable	unstable	not defined

3.4 Conclusions & Future Works

During this project a **Localization and Planning System** has been designed and developed.

Regarding the *Localization* part I started by developing the *Extended Kalman Filter*, however due to its instability, observed during the tests, I decided to use the ROS available *AMCL* node, which implements an *Adaptive-Monte-Carlo-Localization* method for solving the localization problem. The usage of this module was not as it is, but some considerations have been made about the quality of the localization, and according to those considerations, a set of parameters have been modified to reach better performance.

Regarding the *Planning* part I used the modules available in the *ROS Navigation Stack*[10]. Also in this case quantitative and qualitative considerations have been made, and based on those considerations, a set of incremental tests have been designed and performed, first in simulation then on a real robot. I can say that from the out-of-the-box parameters, for our particular configuration, the local planner *TEB* outperforms the local planner based on *DWA*. So, a fine-tuning of the TEB parameters has been performed, reaching a level of performance that overcomes the baseline.

About possible *Future Works*, for sure a more in-depth analysis of the *EKF* will be needed to understand whether the instability is caused by the policy used to obtain the lines or not. Moreover, a further investigation of the local planner parameters would be needed to possible achieve better performance, also because the number of tests performed both in real-world and in the simulation are quite limited with respect to the number and range of parameters, but the maximum has been achieved with respect to the time and resources available.

[10]http://wiki.ros.org/navigation

References

[FBT97] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. “The dynamic window approach to collision avoidance”. In: *IEEE Robotics & Automation Magazine* 4.1 (1997), pp. 23–33.

[Fil+20] Alexandros Filotheou et al. “Quantitative and qualitative evaluation of ROS-enabled local and global planners in 2D static environments”. In: *Journal of Intelligent & Robotic Systems* 98.3 (2020), pp. 567–601.

[NW20] Isira Naotunna and Theeraphong Wongratanaphisan. “Comparison of ros local planners with differential drive heavy robotic system”. In: *2020 International Conference on Advanced Mechatronic Systems (ICAMechS)*. IEEE. 2020, pp. 1–6.

[Pit+18] Maximilian Pittner et al. “Systematic analysis of global and local planners for optimal trajectory planning”. In: *ISR 2018; 50th International Symposium on Robotics*. VDE. 2018, pp. 1–4.

[PRB03] S.T. Pfister, S.I. Roumeliotis, and J.W. Burdick. “Weighted line fitting algorithms for mobile robot map building and efficient data representation”. In: *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*. Vol. 1. 2003, 1304–1311 vol.1. DOI: `10.1109/ROBOT.2003.1241772`.

[Rös+12] Christoph Rösmann et al. “Trajectory modification considering dynamic constraints of autonomous robots”. In: *ROBOTIK 2012; 7th German Conference on Robotics*. VDE. 2012, pp. 1–6.

[SNS11] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[Sta20] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Noetic Ninjemys. 2020. URL: `https://www.ros.org`.

[TBD05] Sebastian Thrun, Wolfram Burgard, and Fox Dieter. *Probabilistic Robotics*. The MIT Press, 2005. URL: `https://mitpress.mit.edu/books/probabilistic-robotics`.

[Zhe16] Kaiyu Zheng. *Ros Navigation tuning guide*. Sept. 2016. URL: `https://kaiyuzheng.me/documents/navguide.pdf`.