IRob - Robot Localization and Navigation

Group 19

Beatriz Paulo (102823), Leonor Fortes (102695) and Tiago Valente (100370)

October 26, 2024

I. INTRODUCTION

Important: in order to have the rosbags that we recorded access this link https://drive.google.com/drive/folders/12Hp-9toPC-4f2T2yw4tT2JyUPnE3aj75?usp=sharing.

A. Motivation

Even though, at this point in time, robotics is no longer a new area, we can comfortably say that it is changing several aspects of our life and of society in general. From factory automation to space exploration rovers, these devices have come a long way from Shakey - the first mobile robot that utilized vision to accomplish its tasks. Thus, it is of utmost importance that we understand the very basics of how robots work and behave, such as their localization and navigation.

Note that this course allowed us to explore the very basics of localization and navigation in a TurtleBot WafflePi. These devices have a steep cost, due to their great odometry and the relatively expensive 2D LiDAR sensor, meaning that we had the opportunity to explore these robotics fundamentals in a platform that is very much appropriate to do so.

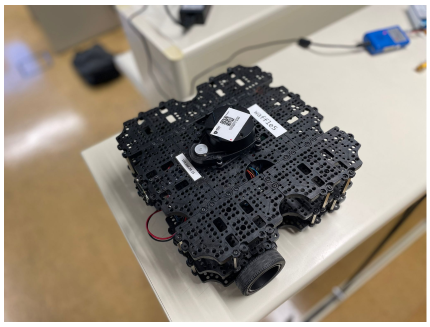

Figure 1: Picture of a Turtlebot3 WafflePi that we utilized in our work.

With the two mini-projects that we have completed for this course, we try to address some of the challenges of mobile robot localization and navigation in "real-world" environments. Although our experiments were all in a specific setting, the aim was to enable the robot to perform well in complex environments with fixed and perhaps even moving obstacles. Hence, the goal is to make the robot be able to accurately and consistently determine its location and navigate toward targets.

In the following section we will discuss what we were asked as the minimum objectives for each mini-project.

B. Problem Statement - Part 1

Mobile robots should be able to reliably estimate their pose (position together with rotation) within an environment, whether it is already known or not. However, in real-world scenarios, the robot's sensors such as LiDAR, odometry or cameras are prone to noise and precision limitations. The fact that the sensor is not perfect, as well as the existence of complex environments, can lead to much higher uncertainty in the robot's perception and consequent localization. However, even in an ideal case there can still be situations where a given sensor is more useful than another (e.g., a camera sensor compared to LiDar in a dark room).

Additionally, the robot's motion model (provided by the odometry) which is a form of relative localization is often subject to error due to wheel slip, drift, and inaccuracies in its internal odometry. These errors accumulate over time, leading to a significant discrepancy between the robot's estimated position and its actual location.

The challenge in this part of the project was to implement an effective localization method that could continuously and accurately estimate the robot's position while compensating for sensor noise and motion errors. Bayesian filters, such as Particle Filters or Extended Kalman Filters (EKF), are commonly used to fuse sensor data with a predicting model based on the uncertainties of each measurement and thus providing a reliable estimate of the robot's pose.

The minimum objectives for this part were:

- map provided with dataset and map estimated from the real robot using gmapping - in RVIZ;
- robot paths (estimated by the robot and estimated from the dataset) for MCL and EKF based localization, respectively, in RVIZ;
- For the dataset only: ground-truth path in RVIZ;
- For the dataset only: error (estimated position minus ground truth position) along the path with associated uncertainty (computed from the estimate covariance matrix) for EKF based localization.

C. Problem Statement - Part 2

Once the robot is able to successfully localize itself, the next challenge is to navigate to a specified target or goal while avoiding obstacles. The robot must not only plan a feasible path, but also be able to follow it reliably while adapting to real-world circumstances, such as unexpected obstacles.

Path planning in possibly dynamic environments presents its own set of challenges. The robot must account for both mapped and unmapped obstacles that can appear at any time. Furthermore, these unmapped obstacles can be static, which should be easier to deal with, or they could be moving. Additionally, the uncertainty in the position has to be low for the robot to confidently navigate the environment but we believe that the AMCL that we are using assures this. Finally, if the robot is able to consistently come up with and complete feasible planned paths, we would like them to be as efficient as possible.

To address these challenges, this part of the mini-project employs Rapidly Exploring Random Trees (RRT) and Dynamic Window Approach (DWA) algorithms for path planning, which are very fast and, with some fine-tuning, good at finding feasible paths to the goal. On top of that, we also present simple solutions to how irregular the RRT plans are and to minimize the distance traveled by the plan.

The minimum objectives for this part were:

- planned paths – in RVIZ;
- actual robot path (estimated by the robot) – in RVIZ; .
- comparative analysis of the planned paths based on a set of relevant metrics.

II. MINI PROJECT 1 - LOCALIZATION

For the 1st Mini-Project, as was mentioned in the problem statement section, we were meant to make the robot be able to correctly localize itself and build a map of its surroundings, using the ROS packages with those same purposes. Hence, it is crucial that the robot is able to perceive the environment and, with this information, to be able to algorithmically determine its pose. Stating the problem in another way, our main objective was for the localization algorithms that we used to have a bounded uncertainty (taking care of precision) while simultaneously having decent accuracy of the robot pose. In this Mini-Project, we were given some files to commence our work[1]: a launch file that plays a given rosbag (or just "bag"), some scripts to fix the time stamps of the topics on the rosbags; but, more importantly, a map together with a bag called "slam_easy.bag" were provided. You can see the provided map and more on the following figure:

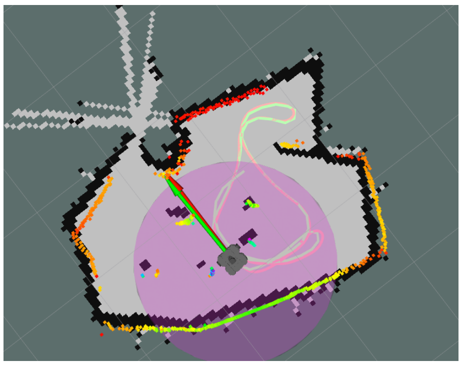

Figure 2: RViz screenshot with the non-corrected EKF for the given dataset and map. **Green path**: mocap data. **Red path**: estimated pose of the EKF. Finally, the circunference centered on the robot shows the uncertainty of the position.

Finally, a bash file called "publish_initial_tf.sh" was provided with the intent of connecting the "mocap" transform to the "map" transform. Instead of calling this bash file every time, we run the "static_transform_publisher" from "tf2_ros" inside our launch files. With this, we are setup to run the provided bag and test our localization algorithms.

Firstly, we were asked to use the "ekf_localization_node" rospackage with the contents of the given bag. This EKF node fuses the odometry and IMU sensor data and, in some cases that we will explore later, even the very precise motion-capture (or "mocap") data that is present in this bag. On the handout, the mocap data is referred to as "ground-truth", however we will just call it "mocap" in order to avoid any confusion with the other term in the literature. Furthermore, as these mocap poses are, in certain parts of our work, being fused with the other sensors' data in the EKF and do not serve only as a comparison (like it was told that ground-truth is supposed to) we believe this name better explains their use.

Note that the paths that are being shown in the previous figure, are being published with an already existent package/script found on the Internet[2].

Put simply, an EKF is a parametric algorithm that fuses the different sensors' data and, together with a prediction of how the state of the robot will evolve, with and without actions, and with the respective process and measurement noise values, it provides an estimation for the state of the robot. The usefulness of this localization algorithm comes from the fact that it is quite general: imagine that we are in an environment that provides high noise measurements, then it will reduce the influence of these measurements on the estimated state and increase the influence of the prediction model. If, instead, the measurements have low noise then the EKF will be highly impacted by them. This will be discussed more later, but we believe that we experienced the first case on the given rosbag. Even though EKF's are very

[1] https://github.com/irob-labs-ist/turtlebot3_datasets

[2] https://github.com/udacity/odom_to_trajectory

useful, due to their parametric nature, we found it challenging to fine-tune these parameters and would like to spend more time on this task to show comparable data to the one we will show for the AMCL algorithm.

Let us start by describing the sensor information and the parameters that we utilized in order to use the EKF node. From the odometry of the WafflePi we are considering the velocities in the x and the y directions (v_x, v_y) directly and from the IMU, as the robot is planar, we are only utilizing the information about the yaw and its velocity (yaw, v_{yaw}). In the cases where we use the mocap data from the provided bag, we are also providing the EKF with its 2D position and with its yaw (x, y, yaw). In our EKF we made some choices regarding the parameters that control this node (still not the ones concerning the noise and uncertainty), such as: the "frequency" was set to $20Hz$ as it is lower than the rate with which the odom publishes its data; "two_d_mode" was set to true as the robot is planar (this, among other things, sets the variables that are not relevant in 2D to be null).

Let us first show the results for the EKF without fusing the mocap data (we will call it the "no corrections case"). It is important to note that to get these following plots we have a script called "publish_position_error.py" which is called inside the "proj1_bag_EKF.launch" file together with rqt_plot.

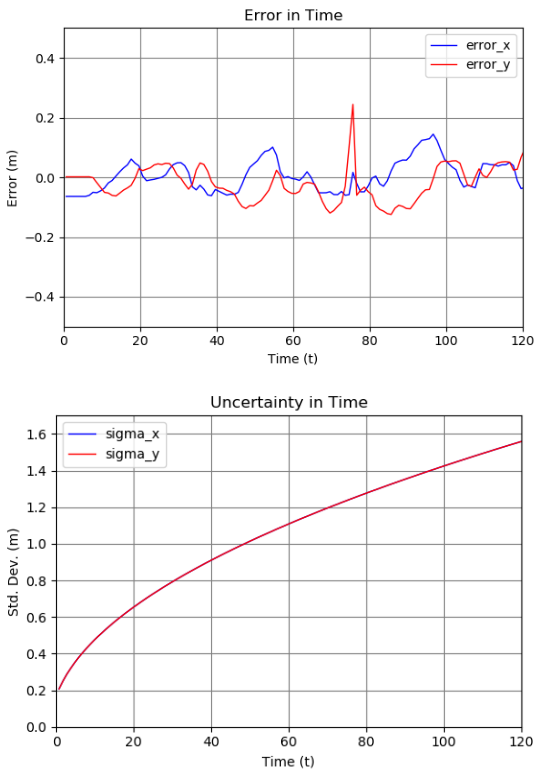

Figure 3: **Above**: Error of EKF-estimated pose with respect to the mocap data. **Below**: Standard deviation of EKF-estimated pose.

As one can see, while the difference between the estimated and the mocap positions is not terrible, the standard deviation keeps increasing indefinitely. This clearly fails one of our objectives of having bounded precision with our EKF. However, this is the expected behavior as, without the corrections from the mocap data, we are mostly using velocity data from the odometry and the IMU, which means that in the integration of these velocities to the estimated position the error will increase in an unbounded fashion.

To try and fix this problem, and as the Mini-Project 1 handout asked, we are using the mocap data in the "proj1_bag_EKF_corrected.launch", by publishing a topic with the transform data between "/map" and "/mocap_laser_link" with a frequency of 1 Hz (as this is the frequency which is asked) with the script "publish_mocap_tf_stamped.py". Afterwards, we fuse this data inside the EKF as we have described before and the results are the following:

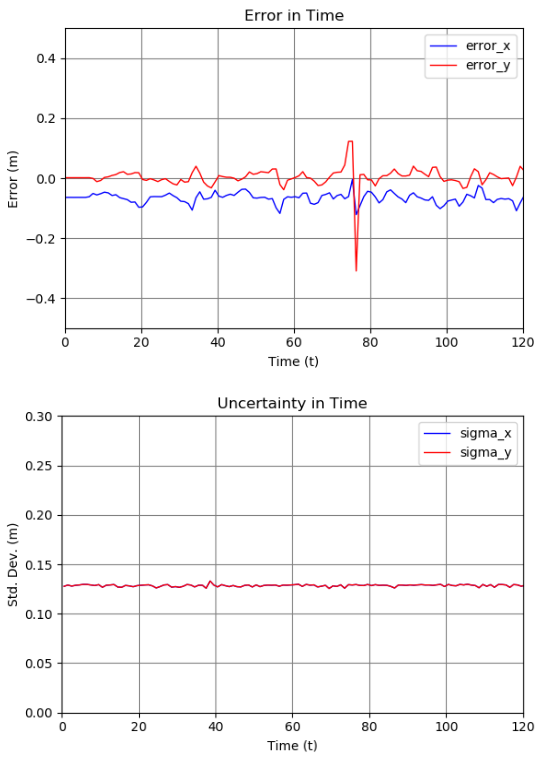

Figure 4: **Above**: Error of EKF-estimated pose (with corrections) with respect to the mocap data. **Below**: Standard deviation of EKF-estimated pose (with corrections).

Thus, we have fixed our issue of the unbounded uncertainty with the EKF. However, looking at the errors in the x and y positions, we do not observe that significant of an improvement. Not to say that the errors are huge, but we would have liked to see some improvements with the mocap corrections.

Finally, let's talk about the different parameters for the EKF node and which values we are using in order to obtain these results. There are two covariance matrices that affect the EKF localization node: the "process_noise_covariance" and the "initial_estimate_covariance"; and if one does not specify one of these matrices, it will default to the ones described in section 3.4.5 of [1]. As this algorithm makes use of the Markov Assumption, any measurement of one of the degrees of freedom of the robot only depends on that same measurement in the previous state. Thus, the covariance matrices are diagonal and these entries correspond to the variance of the respective coordinates. We tested several sets of parameters, but the ones that we are using on the rest of the reported results and that we had the most success with are the following (we are only showing the ones we changed from the default values).

On the process_noise_covariance matrix, we changed:

$$\sigma_x = 0.02 = \sigma_y. \tag{1}$$

On the initial_state_covariance matrix, we changed:

$$\sigma_x = 10^{-5} = \sigma_y. \tag{2}$$

For the case of the EKF with no corrections, and with these parameters, the value of the variance at the last instant of the rosbag is around 1.5 while, with the default values, it was 2.5. There exists one more parameter in regular EKF implementations where one can control the covariance matrices of the measurement noise, but we did not find how to do this with the "ekf_localization_node".

For a quick interlude, let us talk about the mapping of the room (LSDC4) with the robot and the GMapping package from the navigation stack of the Turtlebot packages. In short, we connected to the WafflePi and used the "turtlebot3_teleop_key.launch" from the "turtlebot3_teleop" package in order to move the robot with our keyboard inputs. By recording a rosbag with all of the topics that were being published by the robot, moving it and then running "roslaunch turtlebot3_datasets proj1_bag_gmapping.launch", we are then able to save the obtained map to a ".pgm" and a ".yaml" file. With that, we completed step 2 of the Mini-Project 1 handout and the results are shown here:

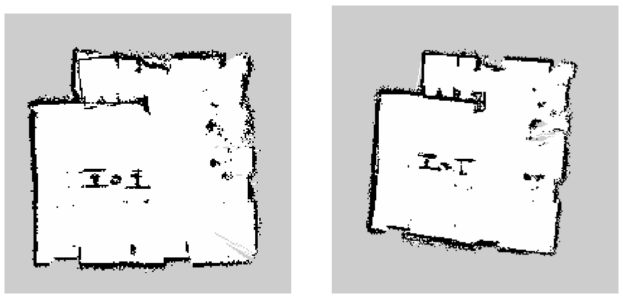
Figure 5: **Left**: Obtained map for run 1. **Right**: Obtained map for run 2.

The other part of this Mini-Project involves another Bayesian filter, however this one is not parametric but instead a particle filter. This algorithm is called "Monte Carlo Localization" but, more specifically, we utilized another version called AMCL, where the "A" stands for "adaptive". The adaptive variant of MCL is slightly more sophisticated as, whenever the particles have converged to the position of the robot, there is no need to have as many particles as we needed in the beginning to fill the map. This way, we need less computations whenever the algorithm has converged. These algorithms use random samples that are successively updated as the robot pose changes until they converge to the actual pose of the robot. Because our attention was mainly on the parameters of the EKF localization we did not change much about the parameters of the AMCL algorithm. In fact, for the results that we are going to show next, the parameters are the default ones (as provided in the turtlebot3_navigation launch files) as we found that these parameters worked great. It wouldn't be one of our group's top priorities, but if we had more time we would certainly pay some attention to these parameters in order to improve the efficiency of this algorithm.

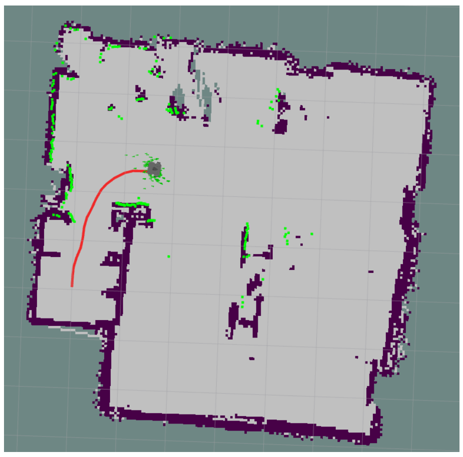
Figure 6: RViz screenshot with AMCL for the map on the right of figure 5. **Green path**: path of the "amcl_pose" data. The small green arrows show different points that the AMCL algorithm is considering as the robot pose.

With this localization method we obtained the following results for the rosbag data:

Looking at figure 7, we can see how the uncertainty starts quite high (around 0.2 m) but quickly converges to a value near zero, while the error seems to be more erratic. However, looking at RVIZ after running "proj1_bag_AMCL_2.launch", one can see that the PointCloud2D clearly converges to the actual position of the robot which is much more in line with what we expect given the uncertainty plot. Running the "proj1_bag_AMCL.launch" also shows how this algorithm works quite well on the actual robot and the mapped room. Because of the clear superiority of this algorithm compared to our EKF, and because of its integration with the navigation stack, we used it for Mini-Project 2.

Lastly, we wanted to compare these 3 localization methods (EKF with and without corrections and AMCL) between one another. Additionally, we also wanted to explore the impact of each method's parameters on their performance. In order to accomplish this task in a quantitative way, we utilized two different metrics:

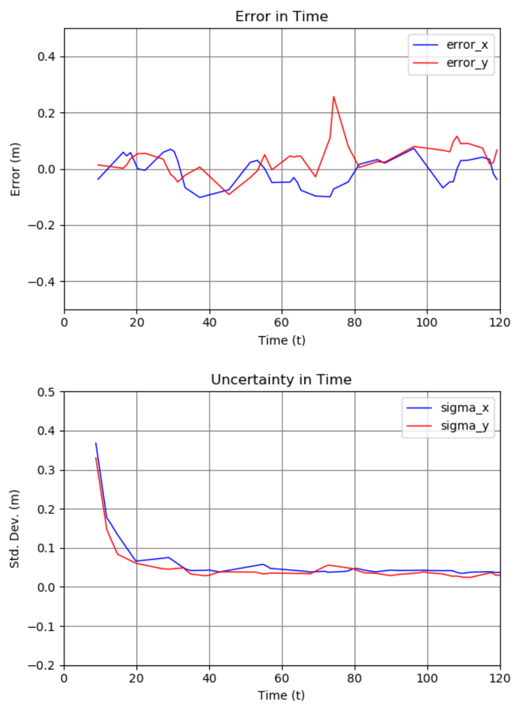

Figure 7: **Above**: Error of AMCL-estimated pose with respect to the mocap data. **Below**: Standard deviation of AMCL-estimated pose. These plots vary slightly between runs due to the random nature of MCL. For $t < 10$ the robot is stopped, that is why there is no data in that region.

- Mean Squared Error (MSE): which is being printed whenever one runs the simulation for one of the cases with the mocap data. The value that is being shown in the table is for the last point that is printed;
- Procrustes Accumulated Error: which was obtained by recording the points of the mocap data and of the estimate of the filter (this was only done for the EKF on the "publish_position_error.py" script) and running scipy's version of Procrustes algorithm in [2] (with the "procrustes.py" script) on this data.

Concerning the MSE, we obtained the following results:

	MSE (m^2)
EKF	0.073
EKF (Corr.)	0.066
AMCL	0.074

Table I: Mean Square Error (MSE) comparison between the methods with the mocap data.

With these results we can confirm what we had stated before: the EKF corrections seem to fix the ever-growing uncertainty but the error is not reduced significantly; in the case of the AMCL we can observe that the mean error squared is actually greater than for EKF. However, with all of the 3 methods we can see that the MSE is quite small, showing that the estimators are doing a good job.

Before we started looking at the MSE metric, we were using a simple accumulated error to compare the different runs. This has many flaws, with one of them being that this error increases indefinitely. Furthermore, we were warned by the laboratory professor that if the path with the mocap data and the one with the EKF-estimated pose are not aligned perfectly, the error between the two at each point will, in general, become bigger with time. Thus, we decided to try and see what the error would be with the Procrustes algorithm. Simply put, this algorithm (after adjusting the means of the paths to be zero) finds the rotation and scaling factor that minimize the error between the two paths and then returns the accumulated squared error over all points. For the EKF with no corrections, we obtained a considerably small value of $6.62 \cdot 10^{-3} m^2$.

With all of the previous results, we have finished every objective from the handout and obtained two schemes that work decently for the localization of the WafflePi. However, in the following work with the navigation stack we will be using AMCL due to its simplicity and good accuracy. During our work for this project, we also thought about the possibility of using Iterative Closest Point (ICP) localization and plugging that pose inside the EKF localization estimator. In the end our group didn't have time to implement what we have described, but it could be an interesting experiment.

III. MINI PROJECT 2 - NAVIGATION

Like it was explained in the problem statement section, for the 2nd Mini Project we were asked to explore the ROS navigation stack and make the robot be able to traverse a mapped location while avoiding different types of obstacles. In order to do this we utilized two different global planners: one that was already implemented in the "turtlebot3_navigation" package and utilizes the Dynamic Window Approach (DWA) and another one that uses a Rapidly Exploring Random Tree (RRT). For the latter, skeleton code in C++ was provided[3] and the rest was implemented by our group.

Starting from this foundation, we implemented the methods: getNearestNodeId(), createNewNode(), sampleRandomPoint(), extendTree() and inFreeSpace(). Like one can see in our source code, our implementation for every one of these methods apart from the last one is quite straight-forward. The last method, "inFreeSpace()", is supposed to return true if a given position of the map is considered "FREE_SPACE" - meaning that there are no obstacles - and false otherwise. Note that the information about the obstacles at each point is present on a structure called "costmap" [4]. The

[3]https://github.com/irob-labs-ist/rrt_planner
[4]In fact, there is a local planner and a global planner costmap

"inFreeSpace()" method then receives a position in the world frame (e.g. the "map" frame), converts this position to grid indices on the global costmap and, finally, it checks if the value at that grid cell is below (or equal to) a certain threshold.

```cpp
bool CollisionDetector::inFreeSpace(const
    double* world_pos)
{
    unsigned int OBSTACLE_THRESHOLD = 140;
    unsigned int map_x, map_y;
    costmap_->worldToMap(world_pos[0],
            world_pos[1], map_x, map_y);

    auto cost = costmap_->getCost(map_x,
            map_y);

    if(cost <= OBSTACLE_THRESHOLD)
        return true;

    return false;
}
```

Listing 1: Code for our implementation of the "inFreeSpace()" method.

We called this new parameter "OBSTACLE_THRESHOLD" and upon experimentation we determined that the value of "140" works pretty well with our other choices of parameters and costmap that we obtained on the 1st project. With these choices, our RRT planner does not provide plans that overlap with obstacles. Furthermore, the value of the costmap for a cell that is considered an obstacle is "254" and 0 for a "free" cell (the value 255 is reserved for "unknown" cells). Thus the value that we landed on makes sense as a middle ground. In future work, it could be interesting to change the costmap_2d cost curve in order to further optimize this parameter and, consequently, a better avoidance of obstacles by our planner since it is able to avoid obstacles, but in some cases it gets too close for the local planner.

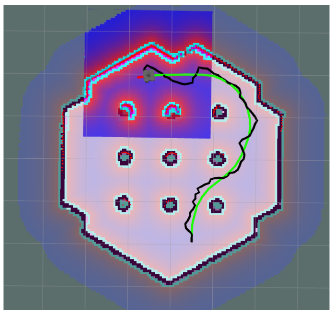

Figure 8: RViz screenshot with the RRT plan. **Black**: RRT provided path. **Green**: predicted path from "amcl_pose". One can also see the global and the local costmaps.

For a quick note about the parameters that we changed in the case of the RRT planner and that are present in the "rrt_planner/config" directory: we decreased the step size of the RRT algorithm to 0.10 (instead of 0.20) in order for it to be more precise and, consequently, we increased the minimum number of nodes in the tree to 500 (instead of 200). Furthermore, in the "configmove_base_params.yaml" file we also changed the "planner_frequency" parameter in order for the robot to stop asking for plans whenever it has already found a solution. This was especially important due to the robot needing some time to reach the goal yaw after it reached the correct position. Finally, for some of our experiments with the robot using RRT, we also increased the "planner_patience" parameter such that, for more distant goals, the local planner waits longer before starting a spinning behavior.

After having a working RRT (global) planner and, together with a DWA local planner, having it work quite well in the simulation, we decided that we wanted to try something outside the planned objectives which was "path smoothing" as a form of "post processing". The motivation behind this is that the plan provided by the RRT planner is quite "jagged" and even though the local planner avoids this jaggedness partially when moving the robot, we believed that path smoothing could improve the provided plan trajectory and make it slightly closer to "optimal". In our work, whenever we mention an "optimal trajectory" we are referring to a trajectory that minimizes the traveled distance and, therefore, it should be composed of several straight lines (that still avoid the obstacles). Our simple path smoothing algorithm works very much like a digital filter or a moving average over three points,

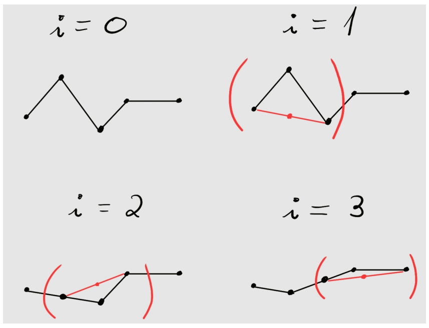

Figure 9: Four iterations of the simple path smoothing algorithm. The final path ($i = 3$) is slightly less jagged than at the beginning ($i = 0$).

As you can see, we iterate over the whole plan considering three points at a time. If we start with the path that is shown in the top left of figure 9, our algorithm takes the first three points and it checks if it can connect i = 0 to i = 2 without hitting an obstacle. If the check returns true (doesn't hit the obstacle), then we set the position of the point i = 1 to be exactly the middle point between the two other ones (we also set the orientation of the plan to be the one that points to the last point we are considering to avoid abrupt changes in direction). In the next iteration we consider the points from i = 1 to i = 3 and do exactly the same until we reach the end of the planned path. At the end, as one can observe on the bottom right of figure 9, we get a path that is clearly smoother and, in general, closer to being optimal (but it is still far from it and the objective of this smoothing is not that).

In order to tackle the issue of the RRT algorithm not giving optimal plans, we have also tried a form of path optimization. Starting from a point that we will call A, we get the first point where the direct path between A and that other point has an obstacle in between. Then, we consider the point before that, let's call it B, and we set the optimized (sub-)path to be a straight line between points A and B. This algorithm can be seen in the following figure:

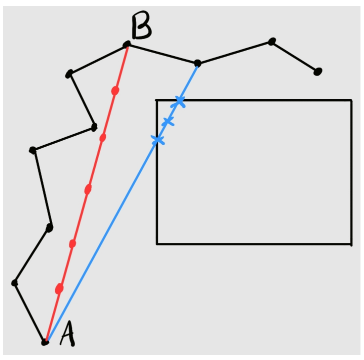

Figure 10: The path optimization algorithm finds the first point which is not able to connect the starting point (in blue we are showing that check). Then, it optimizes the path to a straight line like is shown in the figure. On the next iteration, the starting point will be the end point of the previous iteration.

Another objective for this part of the project was to develop relevant metrics that indicate the performance of the used planners and/or the parameters that we are utilizing for each one. Our group considered these four metrics to be the most relevant ones (note that these scripts are inside the "rrt_planner/scripts" directory):

- Planned Distance[5]: distance (in meters) that the plan spans in order to reach the goal. It is calculated by traveling the plan backwards and summing the distances between nodes;
- Traveled Distance: distance (in meters) that the robot (simulation or real robot) travels as predicted by the "amcl_pose" topic. This calculation is being made on the "calculate_distance.py" script;
- Planning Time: time (in seconds) to get a plan from the global planner. This metric is obtained by measuring the time between the "/move_base_simple/goal" topic being published and the "/odom" telling us that the robot is moving (done in "plan_execution_time.py");
- Total Time: time (in seconds) that the robot takes to come up with and complete a plan to the goal. This is done in the "total_time.py" script and it is the time between the "/move_base_simple/goal" topic being published and the goal being reached (the topic "/move_base/status" has a status equal to 3, meaning success).

Note that due to the random nature of RRT, the metrics oscillate significantly between runs for the same start and end poses. Thus, whenever possible we have made several runs for the RRT and we present an average as the final result. Furthermore, when we run the rosbags that we obtained with the real robot the metrics still vary between runs but this variation is not very significant[6] on the final results and, for simplicity reasons, they were ignored. As a final note, remember that the metrics that are shown are for our choice of parameters (that we will explain shortly after) and can probably be somewhat improved.

With the first two metrics we can show how close to optimal the provided plan is and how much this value differs from the actual traveled distance shows how much the local planner is intervening in the navigation. With the final two, we can see how fast the planner gets a valid plan and how quickly it takes for the robot to execute this plan.

Before showing the results of these metrics for different simulation and real scenarios, note that whenever we wanted the goal position to be exactly the same between different runs we utilized a script that we made. This script is called "nav_goals.cpp" and is inside the package "nav_goals". It allows us to publish a pose to the "move_base_simple/goal" topic by typing the x, y and yaw as console arguments and it eliminates the slight variance that we would see if we were using the RViz tool with the same purpose.

Now we will present the results that we have obtained during mini-project 2. First, we will show results that were obtained in the simulator and, after that, the ones that we obtained with the actual robot.

The example that we will start with will enable us to compare the performance of the different global planners and their respective interaction with the other components of the navigation stack inside the simulation. In order to reproduce these results you will need to roslaunch: the file "turtlebot3_world.launch" that is present in the "turtlebot3_gazebo" package which launches the simulation world, and either the "dwa_turtlebot.launch" or the "rrt_turtlebot.launch" files (present in the "rrt_planner" package) in order to use the DWA or the RRT global planner, respectively. By doing this, you are presented with the RViz interface which allows you to publish a "2D Nav Goal" which, in turn, will request a plan from the global planner that is being used. Whenever the planner is successful in finding a feasible plan, the robot will start moving to its target pose. On the screen you will see the model of the WafflePi together with the map and the two different costmaps - the local and the global one.

[5]This is only implemented for the RRT planner as we did not find a way to obtain the plan for the DWA global planner

[6]This is most significant for the planning time where this uncertainty can account for around 10% of the metric.

Upon publishing a goal pose (either via RViz or by using the nav_goals executable[7]), one can also see the path for the provided plan and, in red and closer to the robot, the effect of the local planner and its simulated trajectory. Choosing a goal pose of $(x, y, yaw) = (0.5; 1.75; \pi/2)$, we have obtained the following results:

	RRT (Average)	DWA
Planned Distance	4.80	-
Traveled Distance	6.85	4.70
Planning Time	1.16	0.22
Total Time	49.71	24.32

Table II: Average metric results for the previously described goal in the simulation.

Now, let us consider this same situation but with path smoothing as in figure 9. As expected, the results for the metrics are very similar:

	RRT (Average)
Planned Distance	5.12
Traveled Distance	6.82
Planning Time	1.21
Total Time	41.00

Table III: Average metric results for the same situation with path smoothing enabled.

However, looking at figure 9 one can clearly observe that the path is smoother and less jagged. This is mostly just a qualitative improvement, as it does not translate to significantly better metrics or to more consistent behavior from the robot. Nevertheless, we decided to include these results in the report as a form of extra experimentation on our part.

Finally, let us consider the results with path optimization on the simulation:

Algorithms	Planned Distance (m)
Regular RRT	4.80
Smoothing	5.01
Optimization	4.05

Table IV: Comparison of the average planned distance results for the same situation with normal RRT, with only smoothing enabled and with only optimization enabled.

As one can see, the distance predicted by the plan is significantly smaller than it was for the two other cases. This means that our algorithm is successful in making the plan much closer to optimal. However, it is important to note that this optimization causes the robot to sometimes get closer to corners of obstacles which may increase the probability of the local planner asking for a new plan.

[7]The executable is inside the "∼/catkin_ws/devel/lib/nav_goals" directory

Moving on to the results that we obtained with the physical robot, it is important to mention that we only have 2 runs for the DWA global planner and 1 for the RRT planner. This limited amount of runs, and consequently small sample size, can be explained by time constraints (if we had more time, we would record more runs) and by the fact that the RRT planner, in its current state, is having some issues in reaching all of the waypoints in the mini-project 2 guide without intervention (meaning that we gave it another goal_pose). However, these issues will be explained in detail later in the report. Even though we have the nav_goals executable to accurately publish goal poses, in these runs the waypoints were published manually using the relevant RViz tool.

Our group recorded a rosbag for each of these runs with the names: "run1_without.bag" and "run2_without.bag" for the DWA and "rrt_bag.bag" for the RRT. In order to run these bags one can launch the following files: "dwa_nav_bag1.launch" and "dwa_nav_bag2.launch" for the respective runs with the DWA global planner and "rrt_bag.launch" for the RRT. In the following table we show the results that we obtained:

Waypoints	RRT	DWA Run 1	DWA Run 2
Start - 1	1.34	0.27	0.26
1 - 2	0.35	0.25	0.24
2 - 3	0.38*	0.25	0.22
3 - 4	0.73*	0.21	0.23

Table V: Planning time results for the runs with the actual robot (without obstacles). Some RRT entries are marked with an asterisk to indicate that the target goal was different.

Furthermore, we also have the total time and the the traveled distance between waypoints:

Waypoints	RRT	DWA Run 1	DWA Run 2
Start - 1	45.50	38.20	37.50
1 - 2	16.69	18.20	15.70
2 - 3	47.99*	42.30	45.41
3 - 4	22.01*	39.51	31.41

Table VI: Total time results for the runs with the actual robot (without obstacles). Some RRT entries are marked with an asterisk to indicate that the target goal was different.

Waypoints	RRT	DWA Run 1	DWA Run 2
Start - 1	8.01	7.73	7.59
1 - 2	2.76	3.14	2.54
2 - 3	8.69*	8.31	8.87
3 - 4	3.70*	5.40	5.78

Table VII: Traveled distance results for the runs with the actual robot (without obstacles). Some RRT entries are marked with an asterisk to indicate that the target goal was different.

Even though neither of the algorithms show horrible planning time, we can comfortably state that the RRT takes considerably longer than the DWA algorithm. In fact, for some of the waypoints, it takes about 5 times as long as the DWA planner, which, depending on the application of the robot, could be unacceptable. However, in terms of the total time and the estimated traveled distance the RRT algorithm (for the single run that we have) is much more competitive compared to the DWA. This tells us that the navigation performance of the two algorithms (with the given parameters) is somewhat close. Although, as we have pointed out before, the last two waypoints can't be compared between all 3 runs as for the RRT the robot did not need to travel as far as for the runs with the DWA. This explains why the RRT seems to be so competitive (or even better) with the DWA on some metrics for those 2 waypoints. In these tests with the robot, we are not reporting the planned distance as, with our current scripts, we would have needed to record it while the real robot is moving (and not with the rosbag). This is clearly an area where with some more time we should be able to solve the issue.

Finally, we tried to make the robot navigate a mapped room with unexpected obstacles. Using only the DWA because its performance is far superior than that of the RRT for these kinds of obstacles, we placed several obstacles around its planned path and recorded the results on the files: "run1_static_obs.bag" and "run1_moving_obs.bag". This global planner was able to successfully navigate the room with unexpected obstacles that were static, but when we switched to moving obstacles the results were much worse. It is important to mention that whenever the robot gets stuck (meaning that the local planner deems the provided plan to be impossible without hitting an obstacle), which happens very often for moving obstacles, the robot starts a behavior where it moves back in seemingly random directions and it only stops whenever it reaches another obstacle and recalculates the global plan. More time would be necessary to investigate the origin of this behavior, but as we did not implement it and it appears in both the DWA and RRT global planners, we suspect that it originates from the default local planner or from the "move_base" package itself. In these tests, we believe that what's more important is to check the behaviors of the robot and its ability to navigate the given environment (and not the actual metrics) due to the much lower success rate of the algorithms. Furthermore, we should mention that the map we were using to run the rrt_planner on the robot was slightly outdated and it had some flaws. For example, one of the cabinet doors was open in this map so the robot had a harder time than it should traveling through that area. One more thing to note is that, occasionally, random points appeared on the costmap and, noting the area where they appeared the most, we believe that the reason for these seemingly random points was the glass door of one of the cabinets in the room. With further fine-tuning, we are confident that we could improve the performance of the robot in these scenarios.

In the end, we were able to successfully navigate a mapped environment with the turtlebot. Thus, we achieved all of the objectives for this 2nd mini-project apart from dealing successfully with moving obstacles and with static obstacles for the RRT algorithm. In the present section, we have pointed out several aspects in which our work could be improved, but to mention a few: we could further fine-tune the considerable amount of parameters that are provided for the local planners, the global planners and the costmap; and we could have recorded more runs with the actual robot.

IV. CONCLUSIONS

This project was a great opportunity for us to dive into the very fundamentals of robotics like the topics of localization and path planning. Regarding Mini-Project 1, we accomplished all the main goals by mapping the room with GMapping, tracking and visualizing the paths of different topics, and showing error analysis in RVIZ. In order to do this, we managed to get both EKF-based and AMCL localization working. Additionally, visualizing the error between the estimated paths and the mocap data helped us see where the robot's estimate deviated from the more precise measurement (as shown in figures 3, 4 and 7). However, it is worth noting that we did struggle considerably with the fine-tuning process for both EKF and AMCL, but the first method is the one where we feel this fine-tuning can improve the results substantially.

In Mini Project 2, we focused on path planning and navigation and we were also able to complete every objective in the handout. Like it was asked of us, we set up a global planner using a Rapidly-Exploring Random Tree (RRT) and, as some extra experimentation on our part, we also implemented some very basic path smoothing and a form of path optimization. With this, the robot was able to navigate through mapped areas and it was able to handle obstacles in real-time when using the Dynamic Window Approach (DWA) global planner as we were not able to do the same with our RRT planner. In order to diagnose these different methods, we built our own metrics, though we realized that there's definitely some room for improvement in how we measure their effectiveness and in what we can conclude from each of these. Furthermore, in the ideal case we would want to see some better results from these planners with our metrics.

Throughout this work we mentioned several ideas to explore in future work on this area. These ideas range from continuing to fine-tune the whole localization and navigation, to implementing totally different methods for localization. The latter refers to our idea of implementing an ICP algorithm that feeds into an EKF. This implementation would definitely bring some challenges, but we believe that the result could be quite interesting. Another idea that we touched upon earlier in this report was the implementation of more sophisticated collision-avoidance strategies that would help the robot handle dynamic environments better. As for some examples of how we would tackle this: we could try to change the behavior where the robot goes in reverse whenever there is a clash between the global and the local planners and, instead of it moving backwards until it would collide again, we could set it to walk back only a set distance. Another interesting idea would be to tackle the costmap curve function.

In summary, this project helped us understand how complex robotics can be and gave us hands-on experience with key concepts on this topic. We learned a lot about the challenges and value of iterative testing, and we're looking forward to tackling even more in the future.

REFERENCES

[1] methylDragon, "Github - sensor fusion with ros." https://github.com/methylDragon/ros-sensor-fusion-tutorial/blob/master/01%20-%20ROS%20and%20Sensor%20Fusion%20Tutorial.md [Accessed: 24/10/2024].

[2] Scipy, "Procrustes - scipy v1.14.1 manual." https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.procrustes.html [Accessed: 24/10/2024].