TÉCNICO LISBOA

Department of Electrical and Computer Engineering

Universidade de Lisboa - Instituto Superior Técnico

Introduction to robotics

2022/2023

Project Report

November 2022

Group: 11

Nico Koltermann - 105653, nico.koltermann@tecnico.ulisboa.pt
Pedro Dias - 96468, pedrondias.work@gmail.com

Contents

1 Introduction

Intelligent systems and mobile robots are becoming increasingly relevant in everyday life, spanning all fields from autonomous driving in traffic, logistics, the military to space exploration or toys manufacturing. Robots are now designed to do dangerous tasks involving heavy-lifting or handling dangerous materials with precision and speeds otherwise impossible.

To keep this precision, the robots must satisfy more and more criteria in accuracy and stability. The robot should react appropriately and intelligently despite sensor noise or unforeseen situations. The subject 'Introduction To Robotics' presents general ideas and solutions for these problems focusing on the handling of sensor data and decision-making for the robot. The students were introduced to studying the various components of a robot and how to program and control it. Specifically, they were tasked with giving a robot one of its most important characteristics, the ability of determining its location in the world and thus being able to move in it.

This paper is a consolidated report of the three mini-projects of the subject: mapping and self-localization, path planning / path following, and decision-making in autonomous mobile robots. They are presented in chronological order of work, the algorithms created in each of the mini-projects, as well as the open-source libraries used to facilitate the work with their respective modifications so suit the project's needs. In each section the results achieved and the respective conclusions drawn are presented.

The result of the project is a robot that is, with some degrees of certainty, able to locate itself in a room, then to identify the goal to be reached and to move towards it, while evading planned and unplanned obstacles in the way.

1.1 Setup: TurtleBot3

To test the code developed throughout the project both virtually and in real life, the robot TurtleBot3, model waffle-pi, and its virtual model in the simulation software Gazebo were used. It features two separate motor-driven wheels and two metallic ball bearings as its way of locomotion, a 360^{o} LiDAR on top, a small battery, a Raspberry Pi computer, a Bluetooth Module, an OpenCR controller (model ARM Cortex-M7) and a front facing raspberry Pi camera which was not used in this project [1].

In order to fulfill the purpose of the project, to create a way for the robot to self-locate and move to a goal-point, the 2D laser scans obtained from the robot's LiDAR were used to both map the room it was in and, after that, also know its current location when comparing the data received to an already made map.

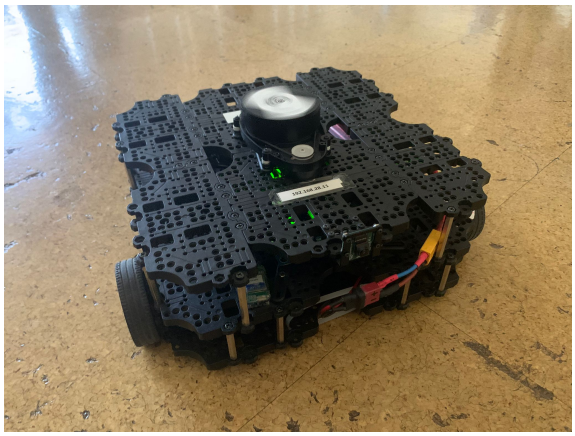


Figure 1: Turtlebot3 model waffle-pi

The code of the project, written in C++ and python, ran on Ubuntu 20 with ROS (Robot Operating System) version noetic [2]. This is a set of software libraries and tools made specifically to help build robot applications which already comes integrated with packages that allow communication, control and checking the status of the robot. To simulate the robot virtually, in order to perform tests on the code without endangering the real robot, a set of pre-made packages were installed, namely TurtleBot3 gazebo [3] and TurtleBot3 simulations [4]. These made a model of the robot in the simulation software Gazebo.

1.2 Setup: Collecting data

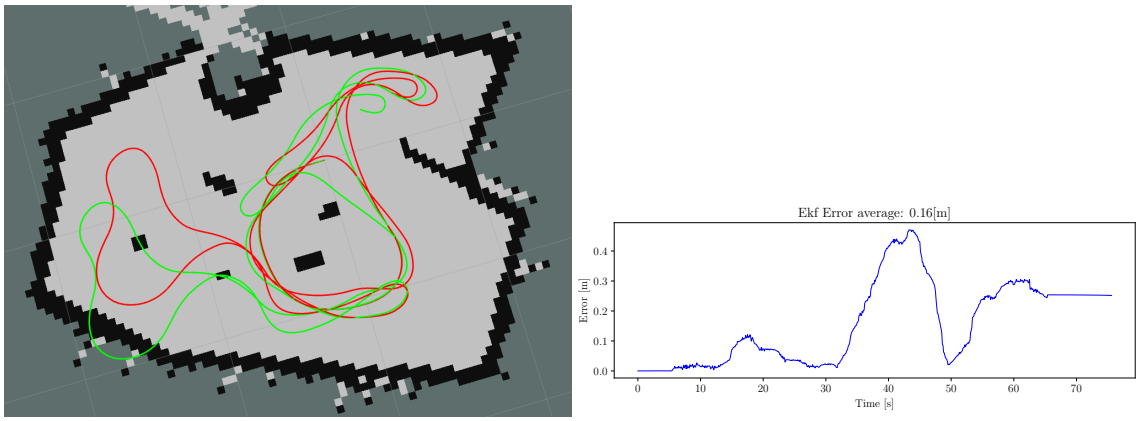
For the simulation runs, the robot URDF [5] was extended by a gazebo plugin. This way, it was possible to read out the ground truth data of the robot and compare it to the localization results. The integrated plugin was libros_p3d [6]. If there is a comparison of ground-truth robot data, it refers to data from ROS, provided by this plugin.
In the following section, it will be described how a robot can create a map, localize itself in it and execute path-planing and path-following. For visualizing the data many rosbags [7] were recorded. To analyze these data and visualize it, the following pipeline was used: After recording the rosbag, a script writes all data of the rosbag into a file, to make it readable and analyzable. After this, a script for the specific case, takes this data and creates visualizations with the matplotlib library [8] in python. After this process, it was possible to save all the visualizations of results as .pgf files to directly include them in this paper.

2 Localization

The first step in the project was to tackle localization, as the remainder of the work stems from the robot having this capability. Localization can be achieved through various methods, which are divided mainly in two categories: relative and absolute localization. Relative localization is achieved through indirect measurements, meaning not directly through measurements of distance to a target or location, but by measuring data such as the rotation of the wheels or the robot's velocity and acceleration, to estimate the new position of the robot relative to its previous one. Absolute localization, on the other hand, is the direct measurement of the distance from the robot to objects or walls, and cross-referencing that data with a given map, the previous data or having a beacon directly on the robot, transmitting its exact location at all times. In this project, both types of localization were used, by using odometry and model matching the 2D laser scans with a map.

2.1 Extended Kalman Filter

The first of the two localization methods is the Extended Kalman Filter (EKF) [9]. The EKF takes the input of sensors, like the odometry of the robot and the inertial measurement unit (IMU), and tries to correct the pose by fusing the sensor inputs and matching them with a mathematical kinematic model. However, as it tries to linearize the problem with only the starting point as a fixed point, it is susceptible to errors in the data received and the model created, leading to growing error over time. This can be seen in the Figure 2.



(a) Path of EKF (green) and robot (red).

(b) Position error of EKF.

Figure 2: EKF run on a rosbag with information of odometry and IMU.

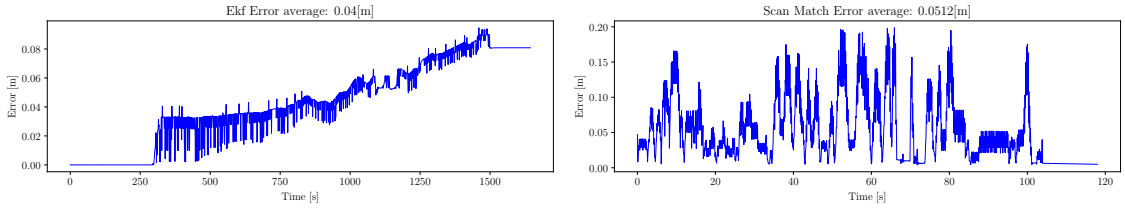
For the implementation, the robot_localization package [10] was used, which provides an implementation of EKF. The data set of the test run in figure 2 a) is provided in the turtlebot3_datasets [11]. For running the rosbag with the algorithm, two state-publishers for transformation (tf [12]) must be included. One transforms from the ground truth frame

/mocap to /odom and the other one from /mocap to /map. It was necessary to fix the tf-tree and to run the algorithms.

At the test run, the position error continuously increases, reaching a peak of 0.4 [m]. After this point, the movement of the real robot intersects with the EKF prediction, causing the error to diminish substantially, but it starts to increase again right after as the intersection passes.

2.2 Scan Matching

For preventing the drift over time of the EKF pose, a scan-match algorithm was introduced, which should correct the error of the EKF calculation. Therefore the Point Cloud Library (PCL) [13] was used to perform an iterative closest point algorithm (ICP) [14] to match the scan to the map. For this, the map and the scan [15] were transformed into the point-cloud datatype of the PCL. By taking the ICP algorithm from the Point Cloud Library, the transformation could be calculated and then applied to the robot position. The resulting robot pose was then published by the scan-matching algorithm. The results of a sample run are shown in the figure 3.



(a) Position error of EKF.

(b) Position error with scan matching.

Figure 3: Position error with EKF and scan matching on the same test run.

It can be observed that the position error of the EKF is increasing over time while the error of the scan matching stays in a constant interval but with a lot of noise. However, as the readings of the scans are noisy, they result in accentuated inconsistencies in the scan match error, which translates to a higher mean error overall.

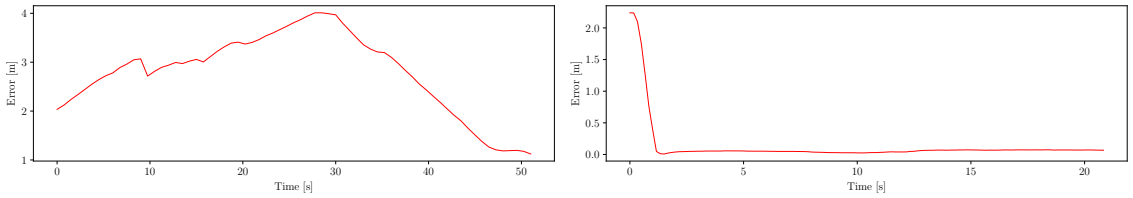
The results of this sample run shows that the EKF is better in short term while the scan matching is needed for more prolonged runs. This conclusion is illustrated in the figure 4, which shows that at the end of the run, while the EKF data has a clear divergence, the ICP is able to keep matching the data with the map and reduce the error.

2.3 Adaptive Monte Carlo Localization

The third localization method is the adaptive Monte Carlo Localization algorithm (AMCL) [9]. It is a particle based filter, which distributes the possible positions of the robot throughout the map and tries to match the scan to the map with ray casting. The best

Figure 4: Point clouds of the Map (green), EKF (red) and ICP (orange)

match gets a higher possibility value. In the next iterations, the points are distributed again, but around the best values of the previous iteration. With these iterations, the best position can converge to the true robot position. For the AMCL implementation, the ROS AMCL package [16] was used. Figure 15a shows a run with AMCL on the turtlebot3 navigation map [3], in which the robot is able to slowly locate itself (< 50 s) with some error (< 1 m). As this package is not optimized for the Turtlebot3, its parameters needed to be changed to adapt the program to the characteristics of the robot (Parameter adapted from [17]). The Figure 15b represents the results after the adjustments, showing a much faster convergence (< 2 s) of the estimated location of the robot to its real location as well as a much smaller error margin (< 0.1 m). For the parameter evaluation, the rqt tools (rqt_reconfigure) were used to reconfigure the parameter during the run time of the algorithm [18].



(a) Before adjustments of AMCL parameter (b) After adjustments of AMCL parameter.

Figure 5: Increase in performance for different parameter on AMCL package.

For a faster and more robust localization, the minimum particles were increased and their update rate disabled (min-particles: $100 \rightarrow 3000$, update-min-d: 0.2 m $\rightarrow -1$, update-min-a: $0.0565 \rightarrow -1$). The standard update rate was based on movement of the robot. However the AMCL should update at every time, as the localization should be independent of the speed of the robot. The initial covariance was increased (initial-cov: 0.5m∗0.5m→ 3.0m∗3.0m). Even if it had a worse initial position, the chance of the robot finding its real position at the beginning was increased as well because it considers a bigger space initially. The result is shown in figure 15b. The algorithm converges faster and finds the robot position in a few seconds. The final parameters are the following:

Table 1: Adjusted AMCL Parameter with most impact on performance.

Parameter	Default	Adjusted
min-particles	100	3000
update-min-d (m)/-a (^{o})	0.2/0.0564	-1.0
odom-alphaX	0.2	1.0
initial-cov (m*m)	0.5 * 0.5	3.0 * 3.0

After a reasonable localization, the kidnapping of the robot was also considered. For this, the robot parameter $odom-alpha$ is important. The higher this parameter, the higher the distribution of the particles and also the uncertainty of the particle cloud on movement and rotation. This means that with movement, the particles get spread around the robot, allowing it to have more adaptability to sudden unexpected changes or errors. This can be seen in figure 6.

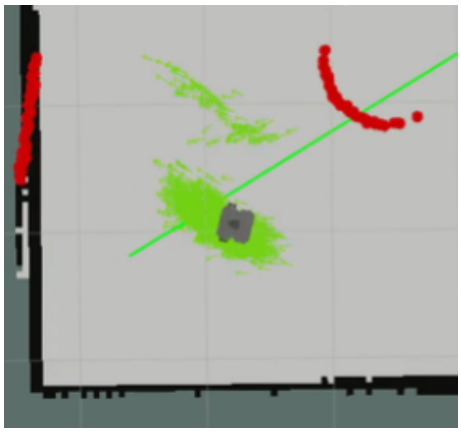


Figure 6: Spread of particles (green arrows) with rotation of robot

During the kidnapping, the localized robot gets a new position and tries again to re-localize. In figure 7 the kidnapping was tested at time = 13 s, by putting the robot in a different position. After 10 seconds the robot got the real position again. The robot re-localized by rotating in the room.

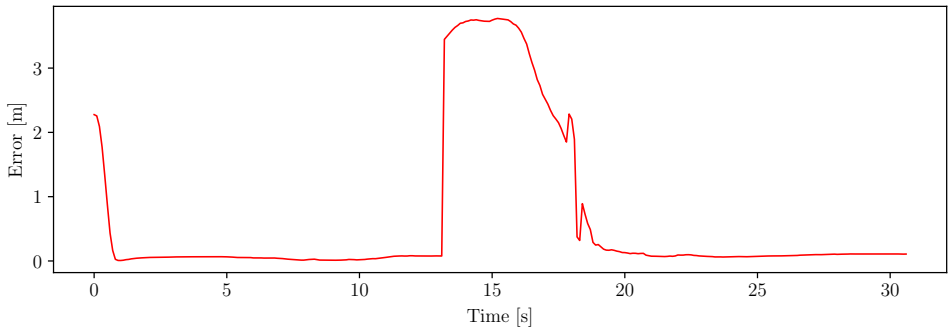


Figure 7: Error of AMCL pose estimation and robot ground truth, with kidnapping of the robot.

3 Mapping

As it was stated before, to localize, the robot needs to compare the data received through its sensors to a map of the room or space it is in. As such, for virtual testing and use in the real robot, a map of the virtual room and the real room respectively was needed. For creating such a map, a SLAM (simultaneous localization and mapping) package for ROS was used, GMapping [19].

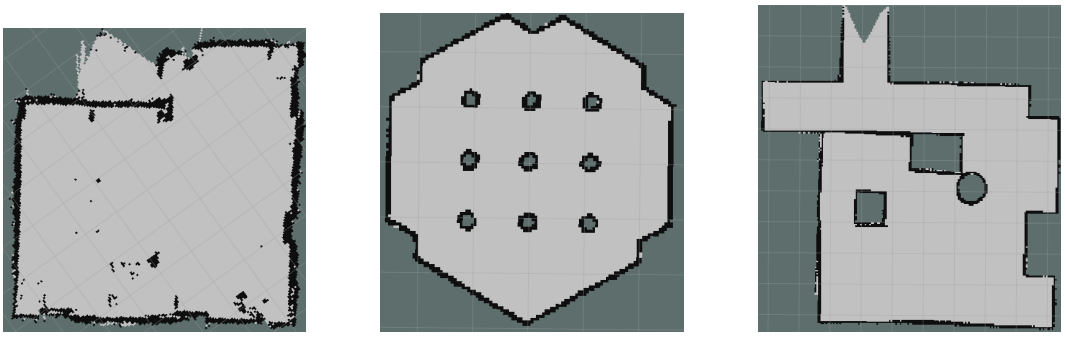
3.1 GMapping

To test the parameters of the GMapping package and to fine-tune them to meet the project's specifications, the world inherited by the TurtleBot3 package [4] was used again. The reason for this is that it already came with a map, making it possible to compare the one created to the already existing one. But as stated before, this world being axis-symmetric made it hard for the algorithm to locate itself. As such, a room in the simulator Gazebo with a shape similar to the real room, and therefore better for testing localization, was created. This was mapped, along with the mapping of the real room.
Similar to the AMCL, the GMapping package also needs its parameters adjusted to suit the specifications of the robot. After the adjustments of parameters, a slow movement was still needed to produce precise maps. If the robot rotates too fast, then a new layout overlaps the old layout and the robot loses its correct orientation. Therefore the mapping was accomplished with slow movement and rotation (example for failed GMapping in appendix 6 figure 17).

Table 2: Adjusted GMapping Parameter with most impact on performance.

Parameter	Default	Adjusted
max-Urange (m)	80.0	3.0
minimum-score (pt)	0	50.0
TemporalUpdate (s)	-1.0	0.5

The parameter max-Urange was changed to reflect the range of the laser used (80m→ 3 m). The minimum-score (0 pt → 50 pt) is an important parameter which has a great impact in mapping the room, as it sets a minimum score for matching the scan with the room map created in that point in time, to be reached in order for that scan to be used, preventing errors occurred when the robot is too far from walls or objects. As with the change with the AMCL, the update rate was also changed to allow for timed updates, making it more independent from the speed of the robot (TemporalUpdate: $-1 \rightarrow 0.5$ (where the negative value means disabled)). However, it was not possible to make it completely independent of the robot's speed as it still needed to move slowly to reduce errors.



(a) LSDC4 map. (b) Turtlebot3 map. (c) Gazebo map for simulation.

Figure 8: Maps created by GMapping.

The results of the test maps for localization and path-planning are shown in figure 8. The first map is the LSDC4 map, where every test on the real robot was accomplished. The second map b) is the turtlebot3 map provided by the turtlebot ROS package. The third map c) was added to cover more test cases in the simulation, before testing on the real robot.

4 Path Planning and Motion Planning

After the localization and a successful mapping, the robot now knows the environment and can start to navigate inside it. For the navigation, the robot must first plan a possible path, take care to avoid unforeseen obstacles (not already mapped) and take the most efficient way. Then the robot needs to execute actions calculated by a controller to follow the path. In the following section this procedure is presented on the TurtleBot3.

4.1 Waypoint following

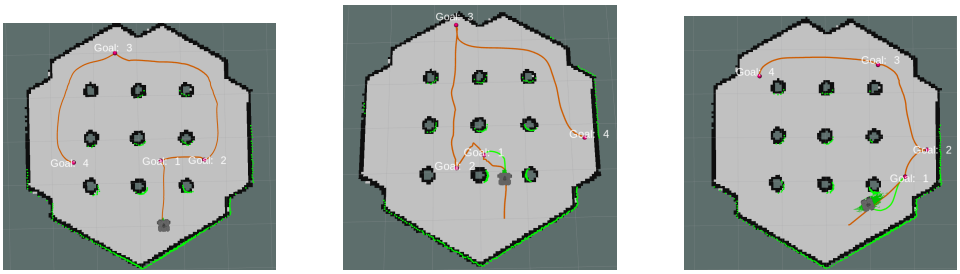
For planning a path, the robot needs one or more goal points. In the project, four waypoints are used to plan a global path. For this, a new node was implemented, where the user can choose between two different types of way point selection:

- Four points input by the user (x, y coordinates). These values are passed by a configuration file to the planning node.
- Selection of four random points on the map. This method only needs a provided map of the environment and no additional information from the user.

The first method just sets the waypoints and passes it to the path-planning. The second method subscribes to the occupancy-gridmap [20], provided by the map server [21]. First, only the empty cells are selected as possible waypoints, then all cells near to a wall also get filtered. Because of the robot size of ~ 30 cm the threshold of selected points was at least 40 cm to avoid collision and provide enough space for the robot to move. Out of these cells, in the end, four points were uniformly and randomly taken and also passed to the path planning. Then a collection of different non-trivial test scenarios were created by this method without any need for knowledge of the map. This can also be applied to the path-following on the real robot in the LSDC4 map.

For the global pre-planning of the trajectory, the *make_plan* service of the move_base [22] was used. The global trajectory starts at the robot's position. Then it calculates all possible paths to the next four waypoints and takes the shortest one. This is done for all four waypoints, until a global path is produced, see the orange path on figure 9. Instead of a greedy approach, a more precise combinatorial approach, such as traveling-salesman problem [23] or a heuristic could be used, but because of a lack of time in the end of the project, no improvement was implemented.

After the calculation of the best order of the waypoints, the next point was sent to the ROS action library [24]. Therefore, the move_base [22] was able to use this point for its calculation. This node runs all the time until the robot reaches the goal. So if there is a new obstacle, which changes the order of the points, a new goal point will be sent to the



(a) Config selection (b) Random selection (c) Random selection

Figure 9: Planned path for four waypoints in different scenarios at the turtlebot3 map.

action-library [24] because the path changed. The full pipeline with the four waypoints and the pre-path calculation is presented in the figure 10.

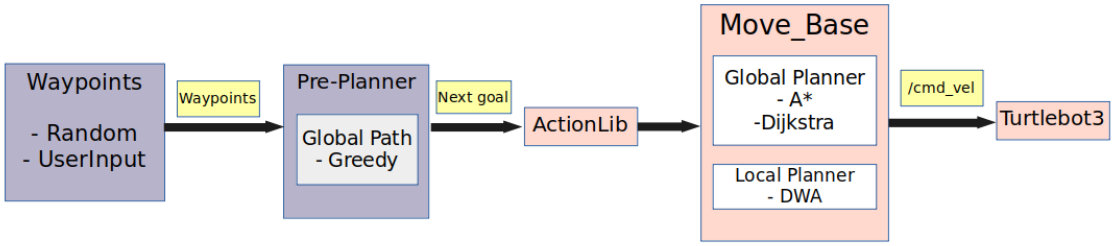


Figure 10: Pipeline describes the path calculation with four points. The pre-path and waypoint calculations were added as new ROS nodes to the move_base [22] pipeline.

4.2 Path following and controls

To follow the path, the ROS package move_base [22] was used. The algorithm has a global planner, which takes the information of the next waypoint, defined previously. For moving the robot, the navigation goal, received by the ROS action library, is used as the next goal. Then move_base calculates a global path to this next point with the A* or the Dijkstra algorithm. To follow this global path, a local planner is used with the dynamic window approach [25]. The local planner uses a costmap, which describes the environment around the robot within weights of reach region. If there is a obstacle, cost will be high, if there is no obstacle nearby, the costs are low. The parameters for this costmap had the most impact on the performance of the controlling and moving the robot. The table 3 shows the changes made for the parameters of the local costmap. The size of this map was increased and also the scaling factor, so the robot tried to avoid the obstacles at a safe distance (defined for this project as the size of the robot, 0.3 m).

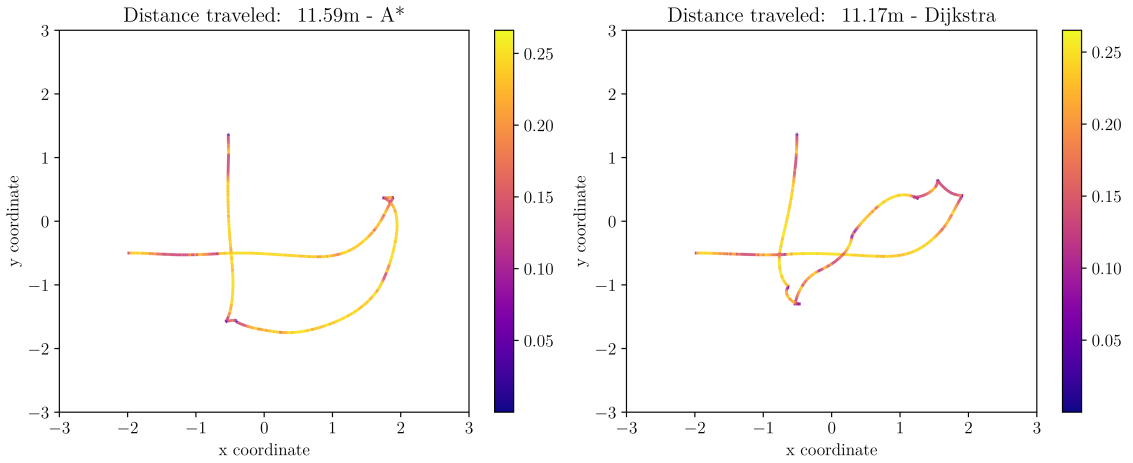
4.3 Quality of path planning algorithms

For testing the two algorithms for global planning, provided by the move_base package (A* and Dijkstra), data of multiple runs was recorded to measure the traveled distance

Parameter	Default	Adjusted
local cost map size	3.0	5.0
cost-scaling-factor	1.0	10.0
inflation-radius	1.0	0.5

Table 3: Adjusted move_base Parameter with most impact on performance.

of the robot. In the test runs, the robot got four waypoints with a defined order, to be independent of other influences and to only show the results, the path traveled, the speed in each point and the total distance. For these test runs the parameters of table 3 were used. The best results can be found in the following figure 11, and the other results can be seen in the appendix.



(a) Path of robot with A* global planning (b) Path of robot with Dijkstra global planning.

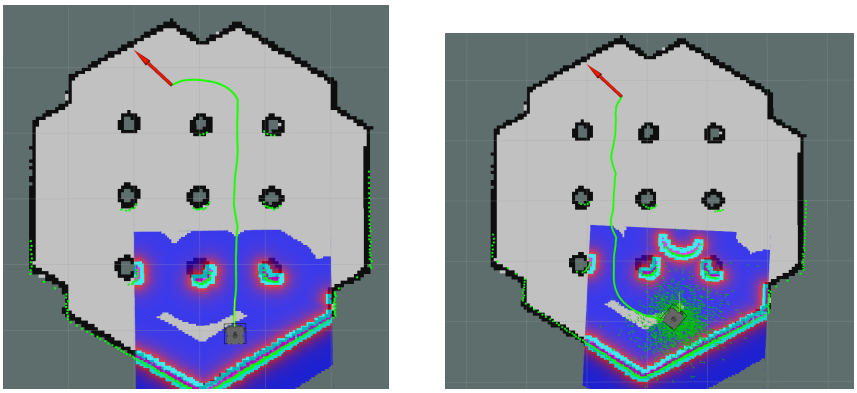
Figure 11: Traveled path of the robot with Dijkstra and A* global planning.

The x and y coordinates describe the position of the robot and the heat-map is the calculated velocity command to the robot. The results show that Dijkstra got, on average, a shorter path for the robot, but sometimes the algorithm has difficulties when there are two possible ways with the same length. Then the robot moved left and right with a uncontrolled movement, because the ways changed all the time. The A* did not have this problem, and the calculation of the path was very consistent over the test runs. In conclusion, the Dijkstra lacked consistency in the calculation, but in the results had the shortest traveled distance of the robot. The A* was consistent in its calculation, but needs more computational power because it is more complex than Dijkstra.

4.4 Obstacle avoidance

The obstacle avoidance was also tested on the real system. The robot input is one goal point and then it calculates the path. Then an unknown object was placed on the map. The robot should change the path to avoid this obstacle. Several tests were done on the

real robot at the LSDC4 map, and the robot was able to circle around objects such as a trash bin or a bag with different placements in each run. One example run for inserting an unknown obstacle and the adjusted path on the turtlebot3 map is presented in figure 12.



(a) Initial planned path (b) Changed path with obstacle

Figure 12: Test for obstacle avoidance on the turtlebot3-map.

In this run, first the robot plans the global plan (with A*) in figure 12a with the previous presented parameter. Then, a sphere was inserted in the gazebo simulator to add an unknown obstacle to the map. Immediately after inserting the new obstacle, the planner changed the global plan and the robot avoided the obstacle and followed the path until reaching the goal point.

5 Decision Making and Planning

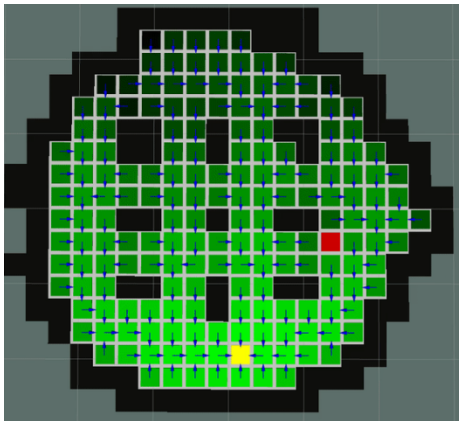
In the previous algorithms, the robot was able to plan a path through a known environment. Now the robot should also include some additional knowledge about the environment and make decisions like if there is a region to avoid, which isn't necessarily an obstacle. In the following, the Markov Decision Process is presented to plan actions for the robot including further limitations than in the previous methods.

5.1 Markov Decision Process

The Markov decision process [9] should plan actions to create and follow a path in a known environment. It takes into account the robot model and defined regions of the environment. If the robot should avoid a location, this has influence in the decision the robot will make.

For the implementation of the MDP a new ROS node was introduced (no other library is used here). The map, converted as an occupancy-grid map, is subscribed from the map server [21]. A lower resolution map was provided at this point, with cells of 30×30 cm. Multiple goal and avoidance points can be selected, the standard rewards are 100 (see figure 13a), similar to [9]. The avoidance point gets the negative reward of the goal point, to be sure to avoid this point with a safe distance. For other occupied points, like walls, obstacles and unknown points of the occupancy-gridmap the reward -10 was assigned. Initially, the reward -1 was applied, similar to [9], but after a few test runs, -10 was determined as better to also avoid the walls in the policy of the MDP.

(a) Initialization of the map.



(b) Visualization of MDP in rViz.

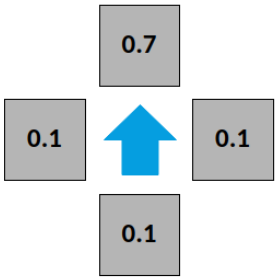
Figure 13: Description of Markov Decision Process

Four actions are possible for the robot in the MDP: forward, backward, left and right. For further understanding of the process, a visualization in RViz [26] was created with the visualization messages [27] (Marker type Cube/Arrow/LineStrip). A green gradient

shows the value from $0 - 100$ and a blue arrow shows the best action in the respective grid. This visualization is presented in 13b.

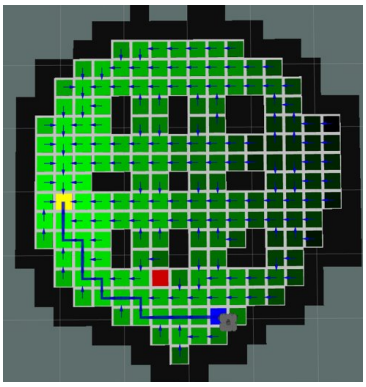
The transition-model should represent the probabilities of the robot actions, therefore the values are selected according to the non-holonomic robot behaviour. Moving backwards means that the robot has to rotate 180 degrees to fulfill the action. Moving forward is only increasing velocity. Just because the turtlebot3 is a non-holonomic robot, the forwards movement gets the most reward while the backwards movement gets the lowest one. As a result, the transition models for the calculation of the best actions and the resulting policies are introduced in figure 13. The probability values have been selected after initial tests and on the basis of their results. These values can be adjusted by the user in a configuration file, loaded on launch of the node. Additionally, three different transition models can be selected, show in figure 14 a) - c). The results of these models are three policies $\{\pi_1, \pi_2, \pi_3\}$ according to figure 14 a) - c), which can be selected by the user in the configuration file of the ROS node. In each value iteration, the best action is selected in each position of the map, until the results converge and an optimized policy is evaluated.

(a) Transition function for π_1

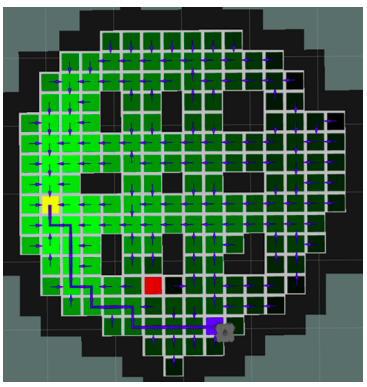


(b) Transition function for π_2

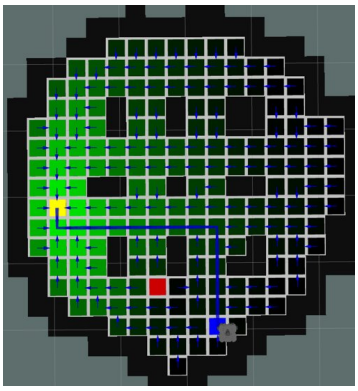
(c) Transition function for π_3



(d) MDP π_1 on turtlebot3 map.



(e) MDP π_2 on turtlebot3 map.



(f) MDP π_3 on turtlebot3 map.

Figure 14: Transition functions for the three policies π_1, π_2, π_3

The results of the MDP with the particular policy is also shown in figure 14 d) - f). The red tile describes the avoidance point, the yellow tile the goal point and the blue tile the robot position. The blue line is the planned path, based on the policy and the best respec-

tive actions in the tiles, starting from the goal point. The more information the model gets of surrounding of the robot, the more weighted is the avoidance point and also the walls. In the first two policies with transition model 14 a) and b) the robot takes the way next to the red avoidance point. But in the third policy, the robot takes a different way, because the negative values of the avoidance point get more weight in the decision and the presence of the avoidance point and wall makes it go around.

For testing the algorithm, the horizon $T = 1$, so the policy was a greedy policy. The selected factor was $\gamma = 1.0$.

5.2 Simple robot controller

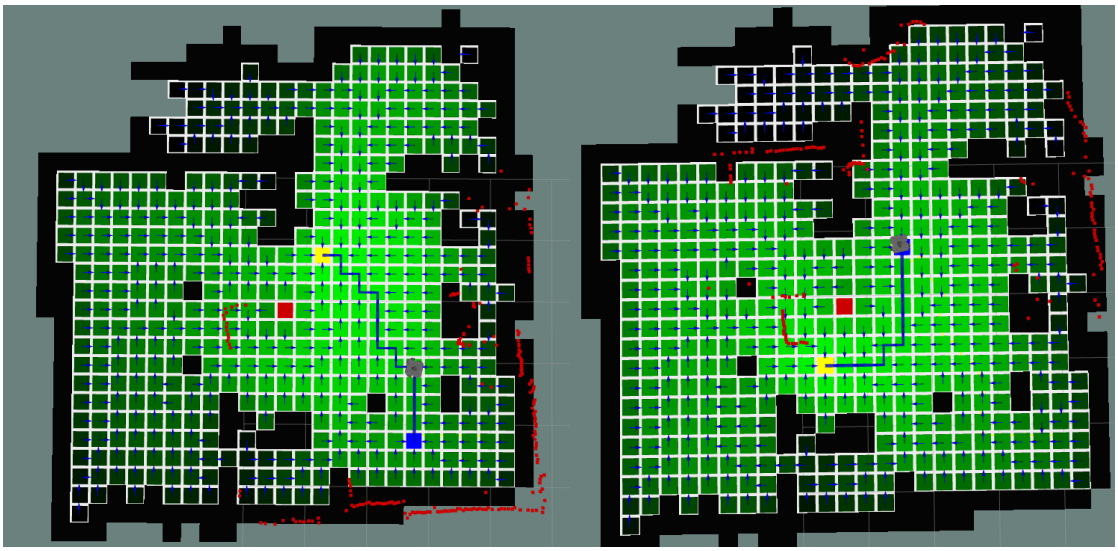
For controlling the robot, a new simple controller was implemented. Because the MDP just calculated a safe way thought the room, there are multiple goal points, close to each other. Therefore a simple controller can be used to follow all these goal points.

Algorithm 1 Simple Controller

$P \leftarrow goalpoints$
while $P \neq \emptyset$ **do**
  while $robot_{x,y} \neq P_i$ **do**
    $\theta \leftarrow \alpha_{robot \rightarrow goal}$
    if $\theta \leq \beta_{threshold}$ **then**
      $cmd \leftarrow rotate_{robot}$
    else
      $cmd \leftarrow moveforward_{robot}$
  $P \leftarrow P \setminus P_i = 0$

As shown in algorithm 1, the robot takes all the points of the calculated path and tries to reach every point after each other in the calculated order. Until all points have been reached, the difference between the point in the room and the orientation is calculated. If the angle is higher then a preset threshold (in the experiment $\beta_{threshold} = 0.1$[rad]), then the robots rotates until the correct angle is reached and it heads in this angle. Afterwards the robot moves forward until the position of the point is reached. This will be done for each point of the planned path until the final goal point is reached.

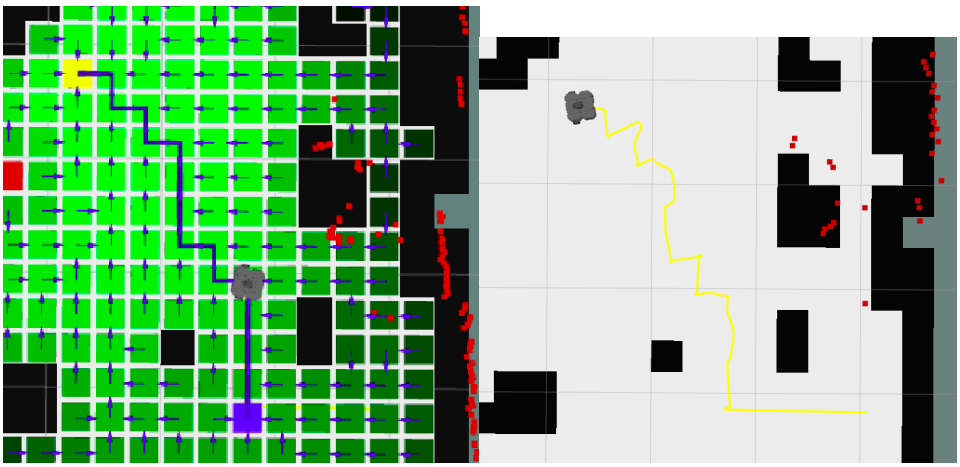
After the simulation tests, the algorithm was tested on the real turtlebot3. For this a new low-resolution occupancy grid map was created by GMapping. Then the robot was placed in the room and localized itself with AMCL package [16]. Two goal points were selected and also one point to avoid. Then the robot calculated the first policy according π_3 in figure 14. The robot follows the calculated line with the simple controller. After reaching the first goal, a new policy was calculated for the second goal point. The both paths and the policies are shown in figure 15.



(a) First goal. (b) Second goal.

Figure 15: Description of Markov Decision Process

The planned path of the MDP and the resulting traversed path of the robot in the LSCD4 test scenario is presented in the following figure 16. The yellow path is the travelled path of the robot on figure 16b.



(a) Planned path with MDP in LSDC4. (b) Traveled path of robot.

Figure 16: Markov Decision Process in LSDC4 map.

Because of the AMCL localization the robot struggles with some noise and had to correct if the estimated position was slightly wrong. But in the end the robot was able to follow the whole path according to the calculated policy without major deviation.

6 Conclusion

In the end the robot was able to localize itself, to build a map of the environment, plan a path and finally follow this path. Developing and testing the software used the advantage of ROS, that changing from the simulation to the real robot was straightforward and so it was possible to work on both systems simultaneously.

The results show the challenge of handling noisy data. A noisy scan can result in an incorrect localization, and an incorrect localization can result in a crash of the path follower. So the methods depend on each other and it begins with the incoming data. It shows, for example, in the scan matching, that increasing consistency with the help of another method leads to a general higher average error. The decision, how to combine algorithms is important and must be chosen very carefully.

However, the project only showed one or two different approaches for the pipeline of localization - mapping - path-planning - controls. The question now is, how to improve performance and precision of new approaches. In the localization the EKF was present and in the mapping the GMapping. But what about a EKF-SLAM [28] or a graph SLAM [29], which combines the localization and mapping part. In the path-planning a controller could be implemented, which would also take advantage of the robot model, like a model predictive controller [30]. It is maybe not relevant in the context of turtlebots, but if robots must fulfill tasks with a high precision, the knowledge about the movement and prediction of movement becomes more important. It would be also interesting to combine the LiDAR input with input of a camera and maybe use object detection. Here one possible and common method would be a convolutional neural network, like the YOLO [31]. But more sensors can also include new problems.

Another aspect beside other approaches is other test setups. Is the robot moving the same, when dark or the robot is in the sunlight, are the results the same? What surfaces will distort the LiDAR data? What sensors can we add or upgrade to improve the performance, or do they just increase energy usage of the robot? These are questions, which can be answered in possible future projects, depending on all the knowledge gathered in this project.

References

[1] *Turtlebot3 Manual*. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. Accessed: 2022-11-10 (cit. on p. 1).

[2] *ROS Noetic*. http://wiki.ros.org/noetic. Accessed: 2022-11-10 (cit. on p. 2).

[3] *ROS Turtlebot3 repository*. https://github.com/ROBOTIS-GIT/turtlebot3. Accessed: 2022-11-10 (cit. on pp. 2, 5).

[4] *Turtlebot3 simualtions repository*. https://github.com/ROBOTIS-GIT/turtlebot3_simulations. Accessed: 2022-11-10 (cit. on pp. 2, 7).

[5] *ROS URDF*. http://wiki.ros.org/urdf. Accessed: 2022-11-10 (cit. on p. 2).

[6] *ROS Gazebo plugins*. https://github.com/ros-simulation/gazebo_ros_pkgs. Accessed: 2022-11-10 (cit. on p. 2).

[7] *ROS Rosbags*. http://wiki.ros.org/rosbag. Accessed: 2022-11-10 (cit. on p. 2).

[8] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55 (cit. on p. 2).

[9] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT Press, 2005. ISBN: 0262201623 9780262201629 (cit. on pp. 3, 4, 13).

[10] *ROS robot localization package*. http://wiki.ros.org/robot_localization. Accessed: 2022-11-10 (cit. on p. 3).

[11] *Turtlebot3 datasets repository*. https://github.com/irob-ist/turtlebot3_datasets. Accessed: 2022-11-10 (cit. on p. 3).

[12] *ROS tf library*. http://wiki.ros.org/tf. Accessed: 2022-11-10 (cit. on p. 3).

[13] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011 (cit. on p. 4).

[14] Zhengyou Zhang. "Iterative Closest Point (ICP)". In: *Computer Vision: A Reference Guide*. Ed. by Katsushi Ikeuchi. Boston, MA: Springer US, 2014, pp. 433–434. DOI: 10.1007/978-0-387-31439-6_179 (cit. on p. 4).

[15] *ROS LaserScan Message*. http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html. Accessed: 2022-11-10 (cit. on p. 4).

[16] *ROS AMCL package*. http://wiki.ros.org/amcl. Accessed: 2022-11-10 (cit. on pp. 5, 15).

[17] *Turtlebot3 navigation package*. http://wiki.ros.org/turtlebot3_navigation. Accessed: 2022-11-10 (cit. on p. 5).

[18] *RQT Reconfigure*. http://wiki.ros.org/rqt_reconfigure. Accessed: 2022-11-10 (cit. on p. 5).

[19] *ROS GMapping package*. http://wiki.ros.org/gmapping. Accessed: 2022-11-10 (cit. on p. 7).

[20] *ROS OccupencyGridMap Message*. http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/OccupancyGrid.html. Accessed: 2022-11-10 (cit. on p. 9).

[21] *ROS map server package*. http://wiki.ros.org/map_server. Accessed: 2022-11-10 (cit. on pp. 9, 13).

[22] *ROS move_basepackage*. http://wiki.ros.org/move_base. Accessed: 2022-11-10 (cit. on pp. 9, 10).

[23] "The Traveling Salesman Problem". In: *Combinatorial Optimization: Theory and Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 527–562. DOI: 10.1007/978-3-540-71844-4_21 (cit. on p. 9).

[24] *ROS actionLib*. http://wiki.ros.org/actionlib. Accessed: 2022-11-10 (cit. on pp. 9, 10).

[25] *ROS DWA planner*. http://wiki.ros.org/dwa_local_planner. Accessed: 2022-11-10 (cit. on p. 10).

[26] *ROS RViz package*. http://wiki.ros.org/rviz. Accessed: 2022-11-10 (cit. on p. 13).

[27] *ROS Visualization Message*. http://wiki.ros.org/visualization_msgs. Accessed: 2022-11-10 (cit. on p. 13).

[28] Abdel Qader and Hisham Fawzi Fayez. "Extended Kalman Filter SLAM Implementation for a Differential Robot with LiDAR". In: 2018 (cit. on p. 17).

[29] Sebastian Thrun and Michael Montemerlo. "The Graph SLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures". In: *I. J. Robotic Res.* 25 (May 2006), pp. 403–429. DOI: 10.1177/0278364906065387 (cit. on p. 17).

[30] Yu-Geng XI, Dewei Li, and Shu Lin. "Model Predictive Control — Status and Challenges". In: *Acta Automatica Sinica* 39 (Mar. 2013), pp. 222–236. DOI: 10.1016/S1874-1029(13)60024-5 (cit. on p. 17).

[31] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. DOI: 10.48550/ARXIV.1506.02640 (cit. on p. 17).

Appendix

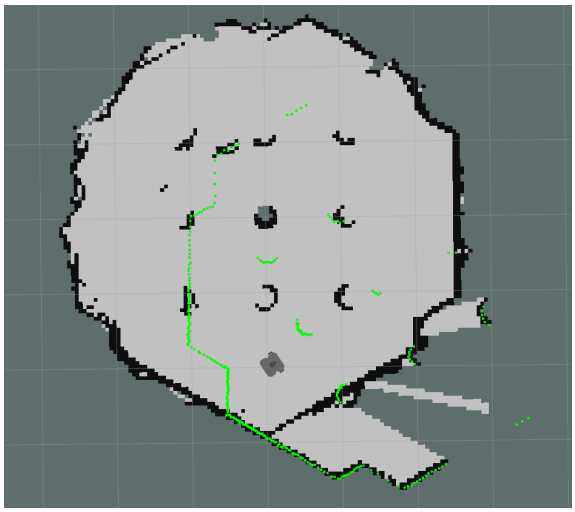
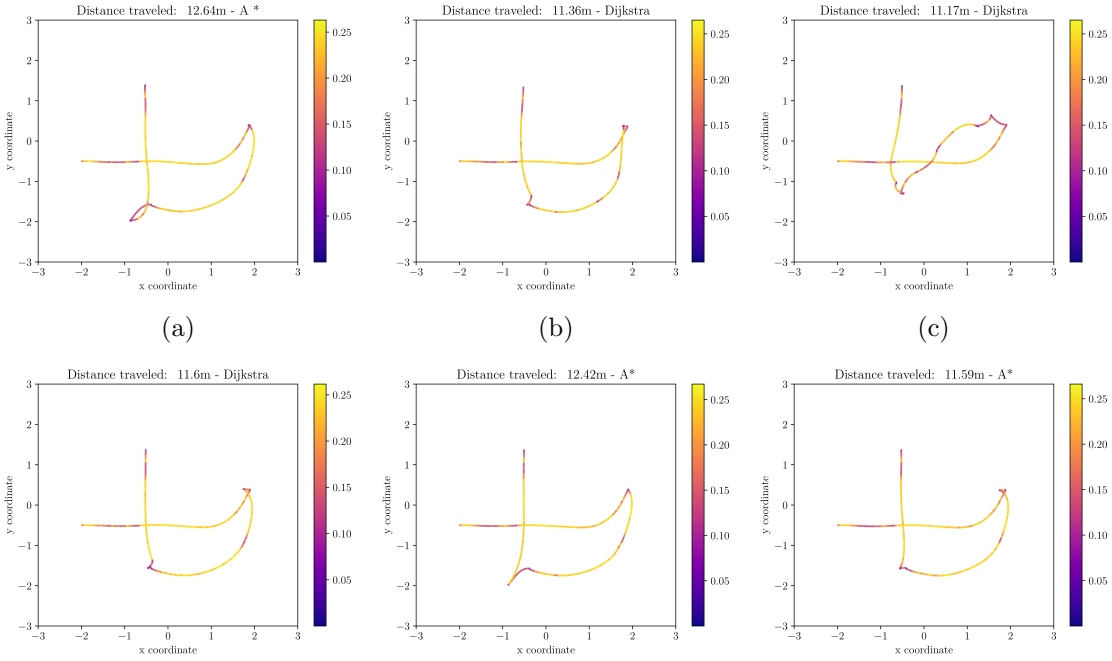


Figure 17: Failed SLAM with GMapping



(a) (b) (c)

(d) (e) (f)

Figure 18: Results of runs with global planner Dijkstra and A*