4TM00 Robot Motion Planning and Control
Group Project - Safe Robot Navigation under Intermitted Localization

Group 5
Devan Sedmak[1](2234238) and Matteo Petris[2](2234203)

***Abstract*— This project designs a safe navigation system for TurtleBot3, combining corrected velocity-based odometry, costmap generation, path planning, and path following. A tailored optimal sampling algorithm plans paths, while a potential field-based follower ensures execution with replanning as needed. Safety is enhanced with costmap margins and limited angular velocity, enabling precise navigation to multiple goals under intermittent localization.**

I. Odometry

In the first part of the assignment we design a ROS node that receives the robot's pose at a low frequency (0.3 Hz) and publishes pose estimates at a higher frequency (10 Hz) to minimize delays in planning and control for navigation. We use the robot's command velocity, scan measurements, and the occupancy map of the environment.

A. Design of the odometry under intermitted localization

The robot in use is TurtleBot3. It is a differential-drive mobile robot with a squared body shape. We can approximate it to a robot with a circular body shape of radius $\rho = 0.22$ m. The equation of motion of the model of the robot is described as a velocity-controlled unicycle motion:

$$\dot{x_1} = v\cos\theta, \quad \dot{x_2} = v\sin\theta, \quad \dot{\theta} = \omega$$

where the state is composed by $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, the position of the robot, and θ, the orientation of the robot, while v and ω are respectively the linear and angular velocity, so the control inputs. There is also a motion constraint that implies there is no sideways motion:

$$\begin{bmatrix} -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = 0$$

We receive the robot's position $\mathbf{x}$ and orientation ω only once every 3 seconds. This means that we need to estimate them every Δt = 0.1 seconds, since that is the update rate for the robot. To do that, we use the Velocity-Based Odometry using Dead Reckoning:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \int_t^{t+\Delta t} \dot{\mathbf{x}}(\tau)\, d\tau$$

$$\theta(t+\Delta t) = \theta(t) + \int_t^{t+\Delta t} \dot{\theta}(\tau)\, d\tau$$

Since the control velocity is received every Δt = 0.1 seconds, it means that it stays constant until a new control velocity is received. So we have:

$$v(\tau) = \bar{v} \quad \text{and} \quad \omega(\tau) = \bar{\omega} \quad \text{for all } \tau \in [t, t+\Delta t]$$

Knowing this we can estimate the position of the robot using the Approximate Position Odometry with the Euler Method:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \bar{v} \begin{bmatrix} \cos\theta(t) \\ \sin\theta(t) \end{bmatrix} \Delta t$$

For the orientation we can use the Exact Orientation Odometry:

$$\theta(t+\Delta t) = \theta(t) + \bar{\omega}\Delta t$$

In this way, we can compute the estimated position $\mathbf{x}(t+\Delta t)$ and orientation $\theta(t+\Delta t)$ of the next state based on the last received state. The next states are estimated iteratively using the previously estimated states.

However, if this process continues, the robot suffers from error accumulation, resulting in significant drift in position estimates. To address this issue, when a new state is received, it is used directly instead of relying on the estimate. This approach significantly improves the robot's motion estimation accuracy.

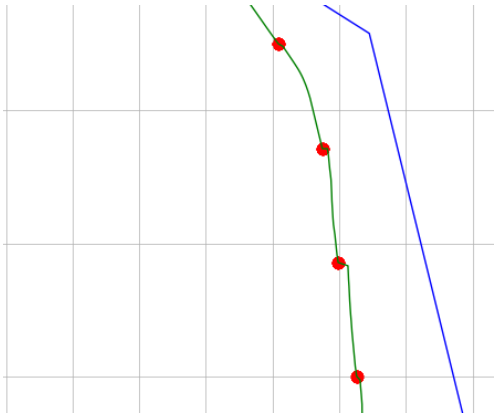

Fig. 1: The green line represents the estimated position based on odometry, while the red points indicate the robot's actual positions. It is evident that there are discrepancies at certain points corresponding to the red markers. These discrepancies occur due to errors in the odometry estimation

B. Limitations of the odometry design

One limitation of this design is that when the angular velocity is high, the estimation accuracy deteriorates, and the robot struggles to perform effectively.

In Fig. 2 the limitations of motion estimation using dead reckoning with a maximum angular velocity of 1.82 rad/s, are clearly demonstrated, highlighting its poor performance.

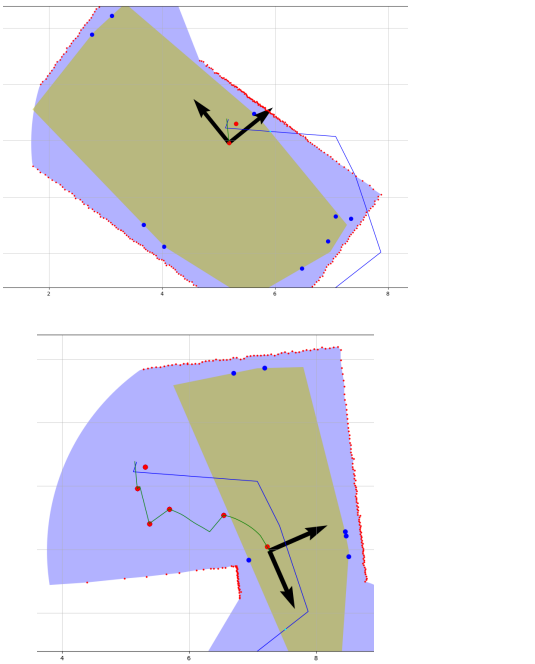

Fig. 2: When using dead reckoning with a maximum $\omega = 1.82\,\text{rad/s}$, the robot struggles to maintain proper orientation within the environment, resulting in zigzag movements

To address this issue, we considered using scan matching when the angular velocity is greater than 0.5 rad/s. The reason is that when the robot rotates in place, the scanned points remain the same but are rotated, making the matching process straightforward.

However, despite our attempts, the scan matching approach was unsuccessful, and the robot was unable to orient itself properly.

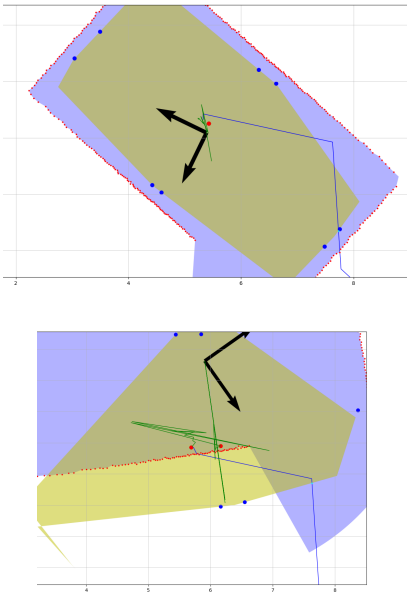

Fig. 3: When using scan matching, the robot can't estimate its position and orientation

This can be seen in Fig. 3, which illustrates the motion estimation performance when using scan matching.

Ultimately, we decided to limit the robot's maximum angular velocity to $0.5\,\text{rad/s}$. In this way, the robot is less prone to estimation errors.

As we can notice from Fig. 4, the motion estimation performance when using dead reckoning with a maximum angular velocity of $0.5\,\text{rad/s}$ is more accurate and does not lead to excessive drift in position and orientation.

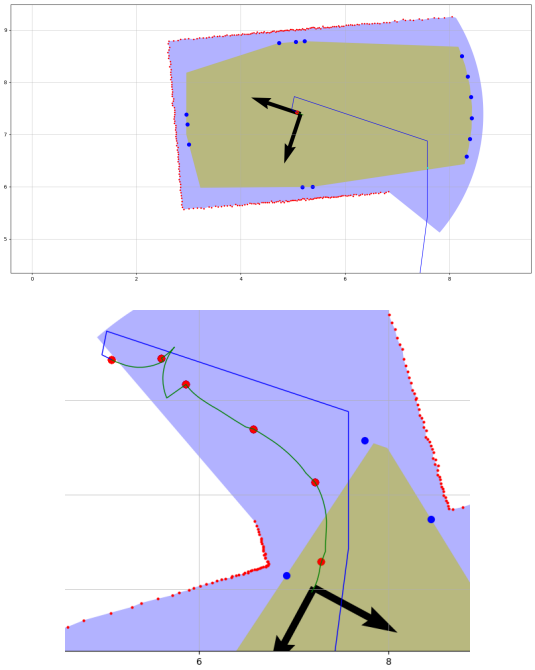

Fig. 4: When using dead reckoning with a maximum $\omega = 0.5\,\text{rad/s}$, the robot rotates slowly and can properly orient itself within the environment. The zigzag movement is limited

II. Costmap

In the second part of the assignment, we design a ROS node that receives an occupancy grid map and generates a safe navigation costmap for reference path planning. We use the robot's scan measurements, command velocity, pose, and goal pose to ensure safe and effective navigation around obstacles.

A. Design of the costmap

The robot is inside a closed rectangular environment with some obstacles, bounded by walls. The environment is represented by the given occupancy grid map. The grid map discretizes the workspace into a grid where each cell represents a portion of the environment.

The occupancy grid map is a 2D array of size 560×380, representing a $28\,\text{m} \times 19\,\text{m}$ area, discretized into cells of $0.05\,\text{m} \times 0.05\,\text{m}$. The map's origin corresponds to world coordinates $(0, 0)$. We use a World-to-Grid transformation to convert from world coordinates (x_1, x_2) to grid indices

(i, j). We also use a Grid-to-World transformation to convert grid indices (i, j) to world coordinates (x_1, x_2).

Each cell's value is assigned as follows:

- 0: The cell is free.
- -1: The status of the cell is unknown.
- 100: The cell is occupied.
- A value in $(0, 100)$: The cell is uncertain, and the value is equal to the probability of the cell being occupied.

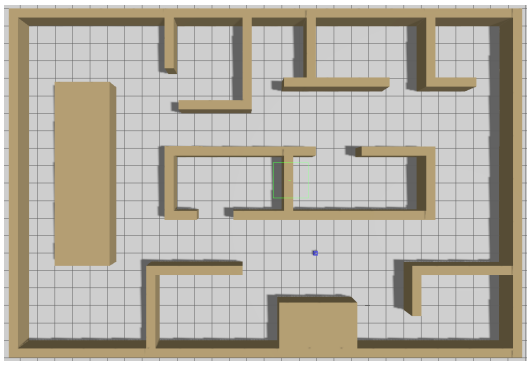

Fig. 5: The environment in Gazebo, the blue object is the robot

Now we construct the map M. It is created from the occupancy grid map definig the cell with value less than a threshold (in our case 1) free, otherwise occupied. If the status of the cell is unknown, we assume it to be occupied.

$$M(i,j) = \begin{cases} 0 \text{ (occupied)} \\ 1 \text{ (free)} \end{cases}$$

Using M we can define the distance map DM as:

$$DM(i,j) = \min_{\substack{(u,v) \\ M(u,v)=0}} \sqrt{(i-u)^2 + (j-v)^2}.$$

Now we impose the $safety\_margin$ equal to the radius of the robot $\rho = 0.22$ (m) and scale it with the resolution:

$$safety\_margin\_in\_cells = \frac{safety\_margin}{resolution}$$

We correct the distance map DM by the $safety\_margin\_in\_cells$:

$$DM'(i,j) = \max\big(DM(i,j) - safety\_margin\_in\_cells,\, 0\big).$$

Finally, we can construct the costmap, called $cost$. It is computed from the distance map DM' as:

$$cost_{ij} = (maxcost - mincost)\, f_{\text{decay}}\big(DM'(i,j)\big)\big) + mincost$$

where $f_{\text{decay}} : [0, \infty) \to [0, 1]$ is the exponential decay function defined as:

$$f_{\text{decay}}(x) = e^{-kx}$$

In particular we select: $k = 5$, $maxcost = 90$, $mincost = 1$.

$mincost$ and $maxcost$ describe the range in which the cost can vary. These numbers are arbitrary; we left them as found in the assignment initial code. An occupied cell has the $maxcost$.

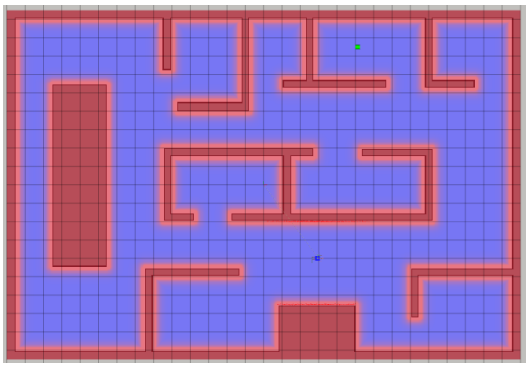

Fig. 6: The computed costmap (red cells are high in cost, blue cells are low in cost). The white dot is the robot's position, the green dot is the goal

B. Limitations of the costmap design

To improve the costmap, we could integrate the laser scan data to update the costmap if some unknown obstacles are detected, if the environment would have been dynamic.

III. Path Planner

In the third part of the assignment we design a ROS node that receives the robot's pose and a goal position, then generates a safe navigation path over an occupancy grid map. The path should guide the robot safely and effectively to reach the goal as quickly as possible. To do that, we use the robot's costmap. It is important to note that it may not always be possible to reach all given goal positions in a complex environment, and executing reference paths that are too close to obstacles can be challenging. Therefore, the objective is to find a reference path with adequate clearance from obstacles; if no such path exists, we indicate that no safe navigation path is available and wait for a new goal.

A. Path planner architecture

A path is a representation of movement strategy. We construct a path as a sequence of segments connecting some grid cells leading to the goal with adequate clearance from obstacles.

We use the Optimal Probabilistic Road Maps (PRM*) algorithm to construct a graph $G = (V, E)$. V is the set of nodes corresponding to the grid cells. E is the set of edges between these nodes. An edge (A, B) is a safe and optimal connection between node A and B. If G is connected, once that the robot's starting position x_{start} and the goal x_{goal}, each already transformed in a grid cell, are also in G, we can obtain a path that connects them.

Before continuing, we have to introduce some concepts. The first one is the weighted posterior sampling ($sample$).

It is a technique used to prioritize sampling points from a costmap based on their associated costs. Importance weights are derived as the inverse of the costs, assigning greater weight to lower-cost regions. This ensures that the weights prioritize more favorable areas. To enable meaningful sampling, the weights are normalized to form a valid probability distribution, with their sum equaling one. Using these probabilities, indices are sampled with preference given to areas of lower cost. Invalid samples with excessively high costs are filtered out and the sampling process is repeated until the desired number of valid points n is obtained. In our case we put $n = 400$.

After we obtain a sample x_{rand}, we directly add it to the graph G. Then we need to find the neighbors of x_{rand} in G. To find the neighbors of a grid cell, we use $neighbor$, a function that finds all the nodes in the graph G that are within a radius from the grid cell. In our case $radius = c\left(\frac{\log(i)}{i}\right)^{\frac{1}{d}}$, where the parameters are $d = 10$, $c = 100$ and i is the number of the current iteration, corresponding to the sample being processed.

Then for each neighbor x_{near} of x_{rand}, we check if there exists a direct safe connection to it ($issafe$). To verify if the connection between x_{rand} and x_{near} is safe, the Bresenham's Line Algorithm is employed. This algorithm identifies the cells that the line connecting the two points intersects, as illustrated in Fig. 7. Once the list of intersected grid cells is obtained, each cell is evaluated for safety by checking if its cost is below a predefined threshold $maxcost'$. Since it needs to be $maxcost' <= maxcost$, we set directly $maxcost' = maxcost$. In this way, it connects x_{near} to x_{rand} safely.

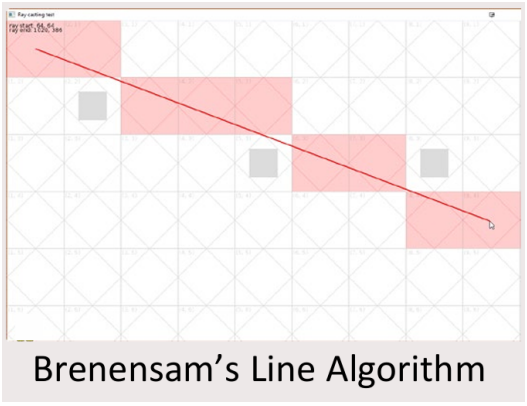

Fig. 7: Visualization of Bresenham's Line Algorithm highlighting the intersected cells along the line connecting two cells

If the connection is safe, we add the edge to the graph. The weight associated to the edge is computed as the local cost (lc) of the edge between a grid cell of a node in G ($startcell$) and a grid cell not in G ($endcell$). To obtain the local cost, we firstly use Bresenham's Line Algorithm to identify the cells ($touched\_cells$) that connect the start cell to the end cell. The mean cost of these $touched\_cells$ is then calculated and combined with the distance between the $startcell$ and the $endcell$ to determine the final local cost, as:

$$lc = \frac{\sum cost(touched\_cells)}{num\_of\_touched\_cells} * ||starcell - endcell||$$

If the connection is safe, this mean is always defined.

We repeat all the steps written above for each sample. Here are all the steps in detail:

Algorithm Optimal Probabilistic Road Maps (RRT*)
Input: Occupancy Grid, Map
Output: Motion Graph $G = (V, E)$

$V \leftarrow \emptyset, E \leftarrow \emptyset$
for $i = 1, \ldots, n$
  $x_{\text{rand}} \leftarrow$ sample(Map)
  $V \leftarrow V \cup \{x_{\text{rand}}\}$
  for $x_{\text{near}} \in$ neighbor$(x_{\text{rand}}, V) \setminus \{x_{\text{rand}}\}$
    if issafe$(x_{\text{near}}, x_{\text{rand}})$
      $E \leftarrow E \cup \{(x_{\text{near}}, x_{\text{rand}})\}$
return $G = (V, E)$

In Fig. 8 we can see an example of a graph G obtained using PRM*.

Fig. 8: Graph representation of an PRM* with 400 nodes.

Once we obtain the graph G, we check if there is a direct safe connection in a straight line from the goal to the neighbors (in a radius of 100 cells) nodes in G of the goal. We do the same for the starting point of the robot. If that is the case, we directly add the goal and starting point and safely connect them to G.

Now we can find an optimal path from the robot's starting position to the goal while minimizing the total navigation cost by using Dijkstra's algorithm.

In Fig. 9 it can be observed that the goal is connected to a node that is not the closest. This approach, which connects the goal to all its neighbors and then applies Dijkstra's algorithm, is effective in minimizing the cost of the path while also increasing the likelihood of finding a valid path. For instance, if the nearest node is obstructed by a wall,

connecting through neighboring node improves the chances of reaching the goal.

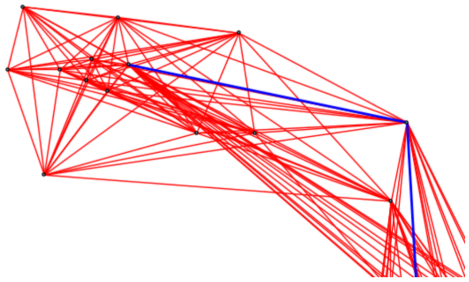

Fig. 9: The part of the graph that connects the goal. In this case the goal is not connected to the very nearest node

Dijkstra's algorithm outputs the sequence of grid cells leading to the goal with adequate clearance from obstacles. This sequence of cells is then converted into the world coordinate system using the grid-to-world transformation. So we obtain a discrete path:

$$p : \{1, \ldots, m\} \to \mathbb{R}^2$$

where m is the number of points in the path. A point in the path is now defined as $p(\tau)$ where τ is the index of the point in the path array.

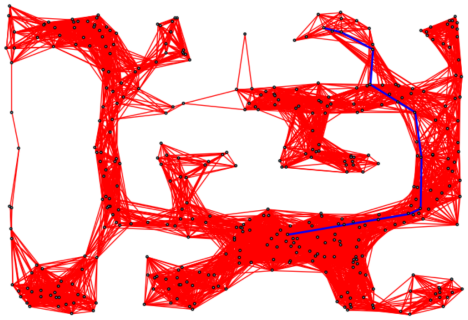

Fig. 10: In blue the selected edges by Dijkstra's algorithm that compose the path from the starting position to the goal over the graph G (in red)

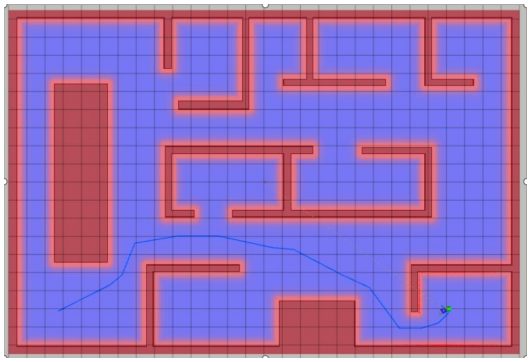

Fig. 11: The path p found by Dijkstra's algorithm from the starting position to the goal over the graph G

The power of this approach is that, even if constructing the main graph is slow, this usually happens only once. Then the graph is only refined, but is not constructed again from scratch. For example, if the robot reaches the goal and a new goal is provided, the new one could not be in the graph. So, like before, if there is a direct safe connection in a straight line from the goal to the neighbors nodes in G, we connect it to G. Otherwise, we reconstruct the graph from scratch, performing $4n$ iterations this time. Then, we check again for a connection and determine if a path exists.

Once we have a safe path, the path follower follows it. But if something goes wrong and the robot moves away from the path too much, there could be no intersections with the provided path and the Local Free Space constructed as in the Path Follower part. In that case the path planner recomputes a new path starting from the newly robot position.

But the new starting position could not be in the graph G. In that case, if there is a direct safe connection in a straight line from the starting position to the neighbors in G, we connect it to G. Otherwise, we reconstruct the graph from scratch, performing $4n$ iterations this time. Then, we check again for a connection and determine if a path exists.

It could happen, even though it is an extremely remote case, that after n iterations the obtained graph is not connected and so there could exist no path connecting the starting position to the goal. In this case we reconstruct the graph from scrach but this time we do $4n$ iterations. Then, we check again for a connection and determine if a path exists. If that is still not enough, then the algorithm indicates that no safe navigation path exists and it returns an empty path. In our case this never happens, because the main graph we construct is dense enough.

B. Limitations of the path planner design

We tried to use a simplified version of the algorithm Informed RRT*. This is a modified version of the RRT*, where we perform a sampling on a costmap with higher density near the goal position. In this case, the importance weights are calculated as the inverse of the square root of the distance from the goal and then normalized to form a valid probability distribution. For each sample we redo the steps from the RRT*, and we add and connect new nodes to the previous graph G.

The Informed RRT* stops when there is a direct safe connection in a straight line from the goal to the closest node in G and adds and safely connects the goal to the graph. In case there is still no safe connection, the Informed RRT* stops after exploring $k = 800$ samples.

In the end, we observed that when a direct connection to the start or goal cell is not established and the Informed RRT* finds a connection, the path remains unresolved because the graph is not fully connected. As a result, we decided to rebuild the graph with more points.

When a node cannot be safely connected to the graph, or the path remains unresolved due to incomplete graph

connectivity, we directly reconstruct the graph. While incrementally adding points to the existing graph could be a more efficient approach to ensure full connectivity or establish a safe connection with the start or goal, we opted for a modular solution. Specifically, we chose to reuse the function that builds the graph from scratch, modifying it to use a larger number of samples for improved connectivity.

IV. Path Follower

In the fourth part of the assignment we design a ROS node that receives a reference path toward a global goal position and generates command velocities to safely follow it while avoiding sensed obstacles. We use the robot's scan measurements, pose, goal pose, and the navigation costmap. It is important to note that it may not be possible to safely follow all reference paths from start to end in complex environments. Therefore, the objective is to follow the path as much as possible. If the path cannot be followed safely, we indicate that replanning is needed (e.g., by printing a message on the terminal). If a new path is planned due to a new goal, our path-following method accommodates this and continues following the new path.

A. Path following strategy

The robot has a 360 2D LiDAR Sensor with a minimum and maximum range of respectively 0.1 and 3.5 meters and an angular resolution of 1 degree. This means that at every scan we obtain 360 samples. This set of samples is called scan polygon and can be seen in Fig. 12.

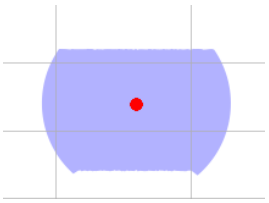

Fig. 12: Scan polygon

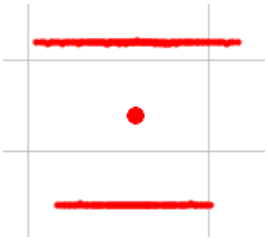

Fig. 13: Scan points

The robot can only detect some points on the surface of the obstacles. These points are called point obstacles (or scan points) and can be seen in Fig. 13. The point obstacles and the distance to a point obstacle with respect to the robot's position $\mathbf{x}$ are defined as follows:

- Point obstacles ($q_i \in \mathbb{R}^2$: Point obstacle position)

$$\mathbf{Q} = (q_1, \ldots, q_l) \in \mathbb{R}^{m \times 2}$$

where l is the number of detected points by the scan.
- Distance to point obstacle q_i

$$d_i(\mathbf{x}) = \|\mathbf{x} - q_i\|$$

Now we need to define the Local Free Space, a region of the environment where the robot can move safely. Firstly we use the defined point obstacles and distance to point obstacle to identify the Local Nearest Points N_Q by iteratively finding the closest point obstacle to the robot's position, adding it to the nearest neighbor set, and removing all points on the opposite side of the hyperplane defined by the robot's position and the nearest point. This process continues until no points remain.

Then, we construct the Modified Local Nearest Points, denoted as N'_Q, which are obtained by shifting the points in N_Q by a distance of ρ along the direction of the distance vector. Next, we take the scan polygon and intersect it with all half-planes that pass through the points in N'_Q and are perpendicular to the distance vector.

These half-planes eliminate the points in the scan polygon that lie on the opposite side of the robot's position. The remaining points consist of the Modified Local Nearest Points N'_Q and a subset of points from the scan polygon. From this subset, we select up to six points (or fewer if fewer than six are available) and denote them as S.

The points in S are then shifted by a distance of ρ along the direction of the distance vector, forming a new set called S'. Finally, the Local Free Space is defined as:

$$LF_Q(x) = \left\{\mathbf{y} \in \mathbb{R}^2 \mid (\mathbf{y} - \mathbf{x})^T(\mathbf{q} - \mathbf{x}) \leq 0 \, \forall \, \mathbf{q} \in N'_Q(c) \cup S'\right\}$$

The $LF_Q(x)$ is constructed as the intersection of all half-planes that pass through the points in N'_Q and S', and are perpendicular to the distance vector. The Local Free Space constructed in this manner has a minimum number of vertices equal to the number of points in N'_Q, and a maximum number of vertices equal to the number of points in N'_Q plus six.

This ensures that even in cases where there are only two, one, or no Nearest Points, an appropriate Local Free Space can still be defined. We provide a comparison between a Local Free Space constructed in this manner and Local Free Space constructed only using N'_Q:

Now we can find the last point of the path that is inside $LF_Q(x)$:

$$\tau^* = \max_{\substack{\tau \in [0,m] \\ s.t. \\ p(\tau) \in LF_Q(x)}} \tau$$

To obtain the projected path goal x_p, we have the following cases:

- If τ^* doesn't exist then this implies that there are no points of the path inside the Local Free Space.
- If $\tau^* = m$ then $x_p = p(\tau^*)$ and this correspondes with the goal.
- If $\tau^* < m$ then x_p is found in the following way

$$d_s = d(p(\tau^*), \partial LF_Q) = \min_{y \in \partial LF_Q} \|p(\tau^*) - y\|$$

$$d_e = d(p(\tau^* + 1), \partial LF_Q) = \min_{y \in \partial LF_Q} \|p(\tau^* + 1) - y\|$$

$$\alpha = -d_e/(d_s - d_e)$$

$$x_p = \alpha * p(\tau^*) + (1 - \alpha) * p(\tau^* + 1)$$

where τ^* is the maximum index of the path within the Local Free Space.

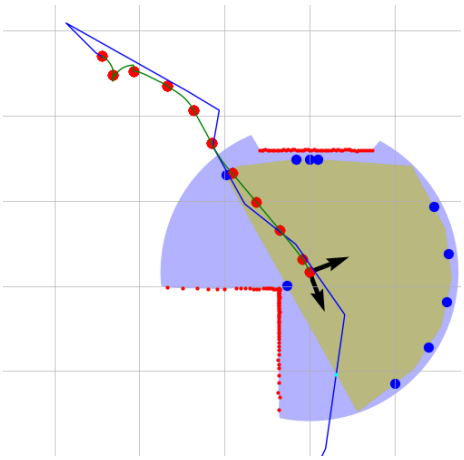

Fig. 14: Example of the Local Free Space (in yellow) constructed using a subset of points (blue points) and the goal path x_p (green point)

Now, we can design a path-following strategy based on potential fields. The key idea is to treat the projected path goal x_p as the center of an attractive field, where the potential reaches its minimum (equal to 0). The attractive field is constructed using the gradient of the attractive potential, which is defined by the Squared Euclidean Distance. This leads to the following formulation:

$$V(x) = ||x - x_p||^2$$

$$\nabla V(x) = 2(x - x_p)$$

Finally, we must choose appropriate linear and angular control velocity gains k_v and k_ω and set the control inputs v and ω equal to:

$$v = -k_v \begin{bmatrix} \cos\theta & \sin\theta \end{bmatrix} \nabla V(x)$$

$$\omega = k_\omega \arctan 2 \left(\nabla V(x)^T \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}, \nabla V(x)^T \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \right)$$

After experimentation, we selected $k_v = 1$ and $k_\omega = 2$. Since these control inputs could exceed the robot's maximum linear and angular velocity, we impose $v \leq 0.26$ m/s and $\omega \leq 0.5$ rad/s (even if the true maximum $\omega = 1.82$ rad/s, following the reasoning in the odometry part).

If the robot comes to a position where there is no intersection between the path and the Local Free Space, then it stops ($v = 0, \omega = 0$) and replans a new path as said in the second part of the assignment. When a new path is provided, it starts to follow it safely.

B. Limitations of the path follower implementation

Using all points for the Local Free Space instead of a subset would improve precision. However, this approach significantly increased computational cost, causing the robot to struggle to adjust in real-time. That's why the Local Free Space we define is constructed using only the Modified Nearest Points and up to six additional points. This approach ensures that the local free space remains entirely within the scan polygon. In our case, the scan polygon is never a complete circle due to the presence of obstacles. For this reason, we limit the construction to six points. However, to ensure the Local Free Space stays inside the scan polygon even when there are no obstacles, we would need to use nine points.

To demonstrate why nine points are necessary, consider the scanner's radius of $R_{\text{scan}} = 3.5\,\text{m}$ and the robot's radius of $\rho = 0.22\,\text{m}$. To guarantee that the local free space remains within the scan polygon, the following condition must hold:

$$R_{\text{scan}} - R_{\text{scan}} \cdot \cos\left(\frac{2\pi}{2n}\right) \leq \rho$$

Calculating for $n = 9$:

$$3.5 \cdot \left(1 - \cos\left(\frac{2\pi}{18}\right)\right) = 0.21\,\text{m} < \rho$$

This satisfies the condition. However, for $n = 8$:

$$3.5 \cdot \left(1 - \cos\left(\frac{2\pi}{16}\right)\right) = 0.266\,\text{m} > \rho$$

This result shows that using eight points does not meet the requirement, whereas nine points ensure the Local Free Space remains correctly defined within the scan polygon.

Additionally, to improve the performance of the robot, we create a new ROS node that is responsible only for visualization of the plots. It subscribes to the topics *pose*, *scan*, *path*, *convex_interior* and *path_goal*. The last two are published by the path follower node.

V. Multi-Goal Navigation

The overall objective of this project is to systematically and properly integrate odometry, costmap, path planner, and path follower into a complete safe navigation framework that can successively visit a finite number of goal positions, published one after another, once the robot is within 0.5 m of the current goal position, under intermitted global localization. We demonstrate that our framework can visit all reachable goal points as quickly as possible. We report how long it takes to visit the three given example goal poses. We also provide additional examples.

A. Demonstration of multi-goal navigation

Our path planner and path follower design successfully ensures the robot visits a finite number of goal positions. The time taken to visit the goals is as follows: 76.88 seconds to reach the first goal, 113.96 seconds for the second goal, and 93.35 seconds to reach the third goal.

The intertime and the total time of the visit of the three provided goals can be seen in Fig. 15. The path and trajectory to the first, second and third goal can be seen respectively in Fig. 16, 17 and 18.

Fig. 15: Goal Publisher node with the times it takes to reach the provided goals

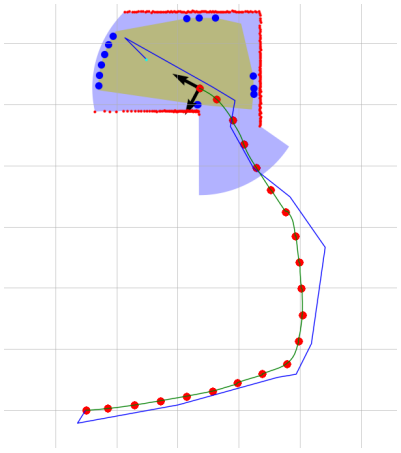

Fig. 16: First goal: the path (blue), the estimated trajectory (green), and the robot's actual positions (red)

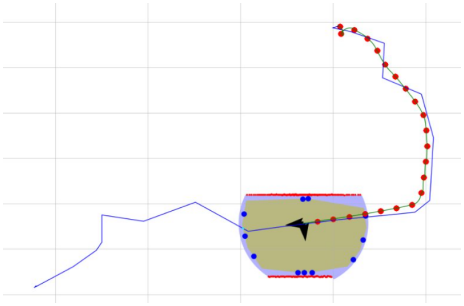

Fig. 17: Second goal: the path (blue), the estimated trajectory (green) and the robot's actual positions (red)

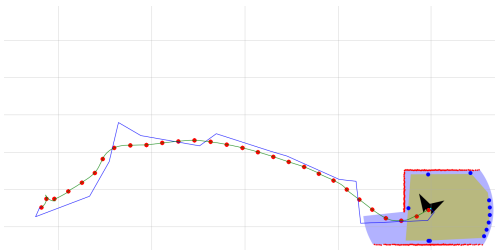

Fig. 18: Third goal: the path (blue), the estimated trajectory (green) and the robot's actual positions (red)

We provide also another 2 example goals (Fig. 19 and Fig. 20), to check if our proposed multi-goal navigation design generalizes also to new goals, even in narrow spaces.

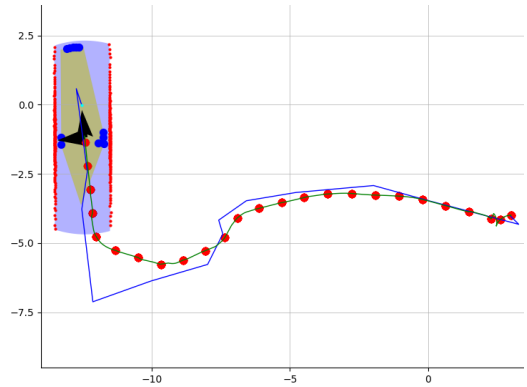

Fig. 19: Goal in [-12.5, 0.0]: the path (blue), the estimated trajectory (green) and the robot's actual positions (red)

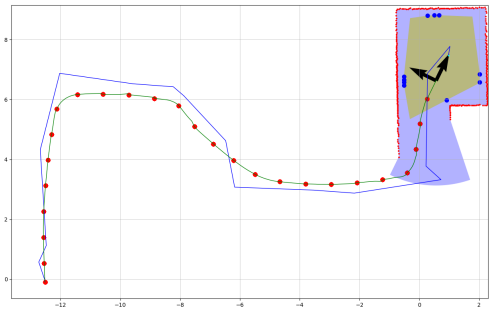

Fig. 20: Goal in [+1.0, +7.5]: the path (blue), the estimated trajectory (green) and the robot's actual positions (red)

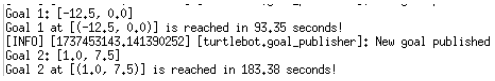

Fig. 21: Goal Publisher node with the times it takes to reach the two example goals

VI. Conclusion

This project successfully developes a safe navigation framework for the TurtleBot3, integrating odometry, costmap generation, path planning, and path following to achieve efficient multi-goal navigation under intermittent localization. By choosing the Optimal Probabilistic Roadmaps (PRM*) over Optimal Rapidly Exploring Random Tree (RRT*), the system gained flexibility for multi-goal scenarios, as PRM precomputes a reusable graph, avoiding the need to rebuild from scratch when the start or goal changes.

The framework demonstrated reliable performance, guiding the robot through multiple goals in complex environments while avoiding obstacles. Although graph construction is initially pretty computationally intensive, PRM*'s reusability significantly reduces planning time for subsequent goals. This approach highlights its effectiveness for dynamic and multi-goal tasks, with potential for further optimization in graph construction and path-following precision.

References

[1] Ö. Arslan, Robot Motion Planning and Control (4TM00) - Course material, Eindhoven University of Technology, 2024.