4TM00 Robot Motion Planning and Control
Assignment 3 - Sampling-Based Optimal Path Planning & Safe Path Following

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

***Abstract*— In this assignment we develop a sampling-based mobile navigation solution using the RRT* and Informed RRT* algorithms to plan safe and efficient paths, comparing it to the previous search-based solution.**

I. Safe Navigation Costmap

A. Introduction

In the first part of the assignment we design a ROS node that receives an occupancy grid map and returns a safe navigation costmap for safe reference path planning. We use the environment map to ensure safety around obstacles.

B. Design of the costmap

The robot is inside a closed rectangular environment with some obstalces, 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 initialized as a 2D array with dimensions 300 × 300, with:

- $width$: 6 meters (discretized into 300 cells along the x-axis).
- $height$: 6 meters (discretized into 300 cells along the y-axis).
- $resolution$: Each grid cell has a size of 0.02m × 0.02 m.
- $origin$: The bottom-left corner of the map is set to world coordinates $(-3, -3)$.

To convert between the grid and world coordinate systems, we implement two transformations:

1) World-to-Grid: The transformation from world coordinates (x_1, x_2) to grid indices (i, j):
$$i = \left\lfloor \frac{x_2 - (-3)}{0.02} \right\rfloor, \quad j = \left\lfloor \frac{x_1 - (-3)}{0.02} \right\rfloor$$
2) Grid-to-World: The transformation from grid indices (i, j) to world coordinates (x_1, x_2):
$$x_1 = j{\cdot}0.02{-}3{+}0.5{\cdot}0.02, \quad x_2 = i{\cdot}0.02{-}3{+}0.5{\cdot}0.02$$

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.

We now construct the binary occupancy grid map B. This matrix is created from the occupancy grid map definig the cell with value less then a treshold (in our case 1) free (False=0) else occupied (True=1).

Now we construct the map M, that is the complementary map of B:

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

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 $\rho = 0.2$ and we 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':

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

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: $k = 0.6$, $maxcost = 90$, $mincost = 1$.

$mincost$ 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 $maxcost$.

C. Limitations of the design

Computing the costmap is computationally intensive because it requires calculating the minimum distance from each free cell to the nearest occupied cell. As a result, the costmap may not always be ready in time for the path planner. To address this, we implemented a validation mechanism to ensure the costmap is loaded correctly before proceeding. Once validated, the path is planned using the path planner node. 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.

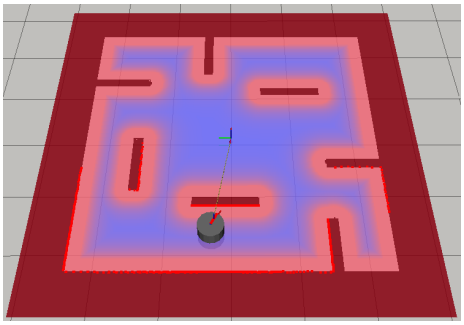

Fig. 1: The cost map

II. Sampling-Based Optimal Path Planning

A. Introduction

In the second part of the assignment, we design a ROS node that receives a goal position and generates an optimal safe navigation path over an occupancy grid map and a costmap using sampling-based optimal motion planning. The path should guide the robot toward the goal while avoiding obstacles and minimizing navigation cost. We use the robot's scan measurements, pose and 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; otherwise, we indicate that no safe navigation path exists.

B. Design of the path planner

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 Rapidly exploring random tree algorithm (RRT*) 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 the node A and B. Since G is a tree, once that the robot's starting position x_{start} and the goal x_{goal}, each already transformed in a grid cell, are in G, we can obtain a path that connects them.

We start constructing the graph by adding x_{start} to it. 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 calculated as the inverse of the costs, assigning higher weights to lower-cost regions, and normalized to form a valid probability distribution. 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 $n = 300$.

After we obtain a sample x_{rand}, we find its nearest node $x_{nearest}$ in the graph. Then we project ($move$) x_{rand} to the Local Free Space of $x_{nearest}$, but constructed using the known obstacles from the costmap and not from the scanner, obtaining x_{new}. If x_{rand} is inside the Local Free Space of $x_{nearest}$, we take $x_{new} = x_{rand}$. To verify if the connection between x_{new} and $x_{nearest}$ 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 the Fig. 2. 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 $maxcost' = maxcost$.

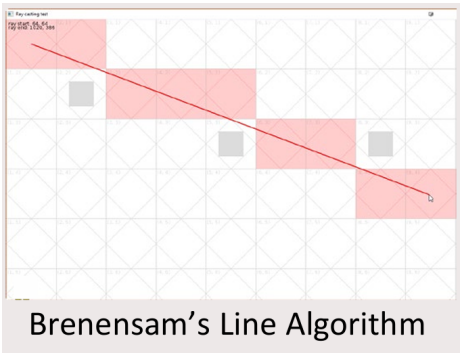

Fig. 2: Visualization of Bresenham's Line Algorithm highlighting the intersected cells along the line connecting two points.

Two other important concepts are the travel cost and the local cost. The travel cost (tc) bewtween two nodes in the graph G is the minimum travel cost obtained using Dijsktra's algorithm. The local cost (lc) between a cell of a node in G ($starcell$) and a grid cell not in G ($endcell$) is computed using the mean of the cost of the cells ($touched\_cells$) that connect the $starcell$ and the $endcell$ using Bresenham's Line Algorithm and the distance between the $starcell$ and the $endcell$:

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

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

We also need to introduce the concept of neighbors of a grid cell ($neighbor$). The neighbors of a grid cell are all the nodes in the graph G that are within a radius. In our case $radius = c\left(\frac{\log(i)}{i}\right)^{\frac{1}{d}}$ where d is a parameter put equal to 10, $c = 10$ and i is the number of the current iteration, corresponding to the sample being processed.

The algorithm then uses these concepts to find the neighbor of x_{new} in G that has a minimum travel cost and a safe connection to it. In this way, it adds and connects x_{new} to G optimally and safely.

The last concept we introduce is the $parent$. The parent of node $x \in V$ is the neighbor with the minimum travel cost to its ancestor (in our case x_{start}) over the graph G:

$$parent_G(x, x_{\text{ancestor}}) = \arg \min_{x' \in V, (x,x') \in E} tc_G(x', x_{\text{ancestor}})$$

The last step of the algorithm uses the concept of the parent to refine the graph by re-evaluating connections for each neighbor of a newly added node. For each neighboring node, it calculates the current minimum cost to reach the neighbor from the start and compares it with the cost of reaching the neighbor through the new node. If the path through the new node is cheaper and safe, the parent of the neighboring node is updated to reflect this new connection. The graph's edge set is then modified by adding the new edge between the newly added node and the neighbor while removing the edge that previously connected the neighbor to its old parent. This process ensures that the graph maintains an optimized and safe structure.

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

Once we obtain the graph G, we check if there is a direct safe connection in a straight line from the goal to the closest node in G to the goal. If that is the case, we directly add and safely connect the goal to G.

Otherwise we 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, then the Informed RRT* stops after exploring $k = 800$ samples.

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.

Algorithm Optimal Rapidly Exploring Random Trees

Input: Occupancy Grid, Map, Start Point x_{start}
Output: Motion Graph $G = (V, E)$

$V \leftarrow \{x_{\text{start}}\}$, $E \leftarrow \emptyset$
for $i = 1, \ldots, n$
  $x_{\text{rand}} \leftarrow$ sample(Map)
  $x_{\text{nearest}} \leftarrow$ nearest(x_{rand}, V)
  $x_{\text{new}} \leftarrow$ move$(x_{\text{nearest}}, x_{\text{rand}})$
  if issafe$(x_{\text{new}}, x_{\text{nearest}})$
    $x_{\text{min}} \leftarrow x_{\text{nearest}}$
    mincost $\leftarrow$ tc$_G(x_{\text{start}}, x_{\text{nearest}})$ + lc$(x_{\text{nearest}}, x_{\text{new}})$
    for $x_{\text{near}} \in$ neighbor(x_{new}, V)
      tempcost $\leftarrow$ tc$_G(x_{\text{start}}, x_{\text{near}})$ + lc$(x_{\text{near}}, x_{\text{new}})$
      if tempcost $<$ mincost $\wedge$ issafe$(x_{\text{near}}, x_{\text{new}})$
        $x_{\text{min}} \leftarrow x_{\text{near}}$, mincost $\leftarrow$ tempcost
    $V \leftarrow V \cup \{x_{\text{new}}\}$, $E \leftarrow E \cup \{(x_{\text{min}}, x_{\text{new}})\}$
    for $x_{\text{near}} \in$ neighbor(x_{new}, V)
      mincost $\leftarrow$ tc$_G(x_{\text{start}}, x_{\text{near}})$
      tempcost $\leftarrow$ tc$_G(x_{\text{start}}, x_{\text{new}})$ + lc$(x_{\text{new}}, x_{\text{near}})$
      if tempcost $<$ mincost $\wedge$ issafe$(x_{\text{new}}, x_{\text{near}})$
        $x_{\text{parent}} \leftarrow$ parent$(x_{\text{near}}, x_{start})$
        $E \leftarrow E \cup \{(x_{\text{new}}, x_{\text{near}})\} \setminus \{(x_{\text{parent}}, x_{\text{near}})\}$
return $G = (V, E)$

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.

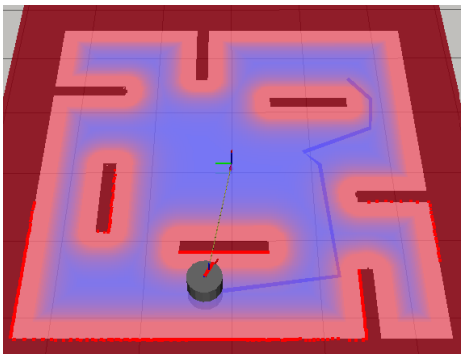

Fig. 3: The path p found by Dijkstra's algorithm from the starting position to the goal over the graph G

The power of this approach is that, even if constructing the main graph is slow, this 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 closest node in G, we connect it to G. Otherwise we use the Informed RRT* biased on the goal and then use Dijkstra's algorithm to find a path.

Once we have a safe path, the path follwer 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 obtained by the scanner. 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. So we redo the same steps as for the goal. If there is a direct safe connection in a straight line from the starting position to the closest node in G, we connect it to G. Otherwise we use the Informed RRT* but this time biased on the starting position. Finally, we use Dijkstra's algorithm to find a new safe path.

It could happen, even though it is an extremely remote case, that a safe connection to the goal or starting position can't be found. In that case the algorithm doesn't search for a path, since it would not find one, but instead it 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 and the Informed RRT* helps to refine it when needed.

C. Limitations

In the case that the robot doesn't find a safe path to follow, it just waits for a goal change or an user input. This limitation could be overcome by either moving the robot a little bit away from the nearest obstacle or projecting the goal to the nearest safe cell. We use Dijkstra's algorithm to determine a new safe path. However, since the graph is structured as a tree, we can opt for a more computationally efficient algorithm specifically tailored to trees, offering better performance

D. Conclusion

Our design represents a compromise between computational cost and optimality, as it significantly reduces the number of nodes in the graph compared to running Dijkstra's algorithm directly on the entire costmap, by constructing a sparse yet effective graph using the RRT* algorithm and refining it with Informed RRT*.

III. Sensor-Based Path Following

A. Introduction

In the third 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 reference path. 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 and we find a new path.

B. Design

The robot in use is RoboCyl. It is a fully-actuated velocity-controlled robot with a circular body shape of radius $\rho = 0.2\,\text{m}$. The model is described with the following equation of motion:

$$\dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt} = \mathbf{u}$$

where $\mathbf{x} \in \mathbb{R}^2$ is the state of the robot indicating its position in the environment and $\mathbf{u}$ is the input command velocity.

The robot can only detect some points on the surface of these obstacles. These points are called point obstacles. 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

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

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.

Now we can define the Local Safe Corridor:

$$SC_Q(x) = \left\{\mathbf{y} \in \mathbb{R}^2 \mid (\mathbf{y} - \mathbf{x})^T(\mathbf{q} - \mathbf{x}) \leq 0 \,\forall\, \mathbf{q} \in N_Q(c)\right\}$$

$SC_Q(x)$ is constructed using the intersection of all half-planes that pass through the N_Q and are perpendicular to the distance vector.

After that we need to find the Local Free Space, defined as:

$$LF_Q(x) = \left\{\mathbf{y} \in \mathbb{R}^2 \mid B(\mathbf{y}, \rho) \subseteq SC_Q(x)\right\}$$

To do that we firstly construct the Modified Local Nearest Points N'_Q, which is obtained by shifting the points in N_Q by a distance of ρ along the direction of the distance vector. The $LF_Q(x)$ is then defined as the intersection of all half-planes that pass through the points in N'_Q and are perpendicular to the distance vector.

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.

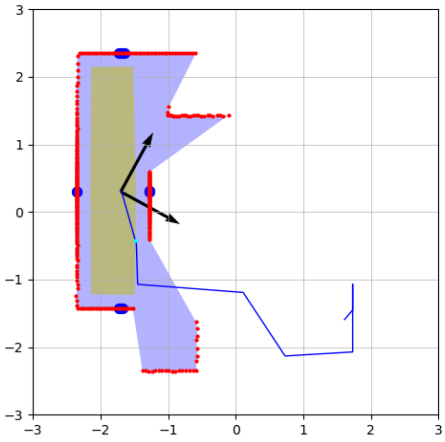

Fig. 4: Example of the Local Free Space and the goal path x_p (green point)

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:

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

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

Now we need to transform ∇V in $\nabla V'$, referring to the orientation angle α of the robot in the environment using this transformation:

$$\nabla V'(x) = \begin{bmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{bmatrix} \nabla V(x)$$

Finally, we must choose an appropriate control gain g and set the control input u equal to:

$$u = -g\nabla V'$$

After experimentation, we selected $g = 1.4$.

If the robot comes to a position where there is no intersection between the path and the Local Free Space, then it stops ($u = 0$) and it 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.

C. Limitations

At first we also used the bounded repulsive gradient jointly with the attractive one. The Modified Local Nearest Points N'_Q were modeled as centers of repulsive fields, where their influence diminishes to zero at their respective centers. But we noticed that, when the robot navigates through a very narrow corridor, sometimes it struggles to follow the path smoothly, as can be seen in Fig. 5. This issue arises because the Modified Local Nearest Points fluctuate significantly, oscillating back and forth. So we chose to remove it and use only the attractive gradient, solving the issue.

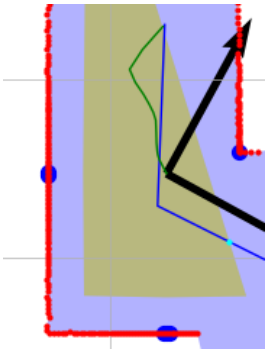

Fig. 5: When the robot is in a narrow corridor and is using also the repulsive field, it tends to oscillate (green line)

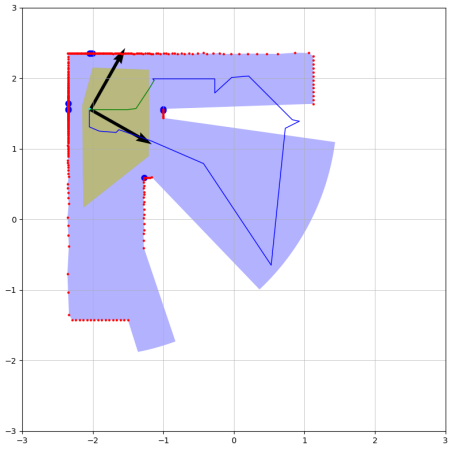

Fig. 6: The robot plans a non-optimal path, but then it finds a shortcut using the Local Free Space

The robot effectively follows the path and reaches the goal in a short amount of time. Moreover, if the path is not globally optimal, the robot sometimes identifies shortcuts by being drawn toward the projection of the goal, as illustrated in Fig. 6. This behavior demonstrates that the path follower partially mitigates the issue of non-global optimality by leveraging the goal's projection to refine the trajectory dynamically.

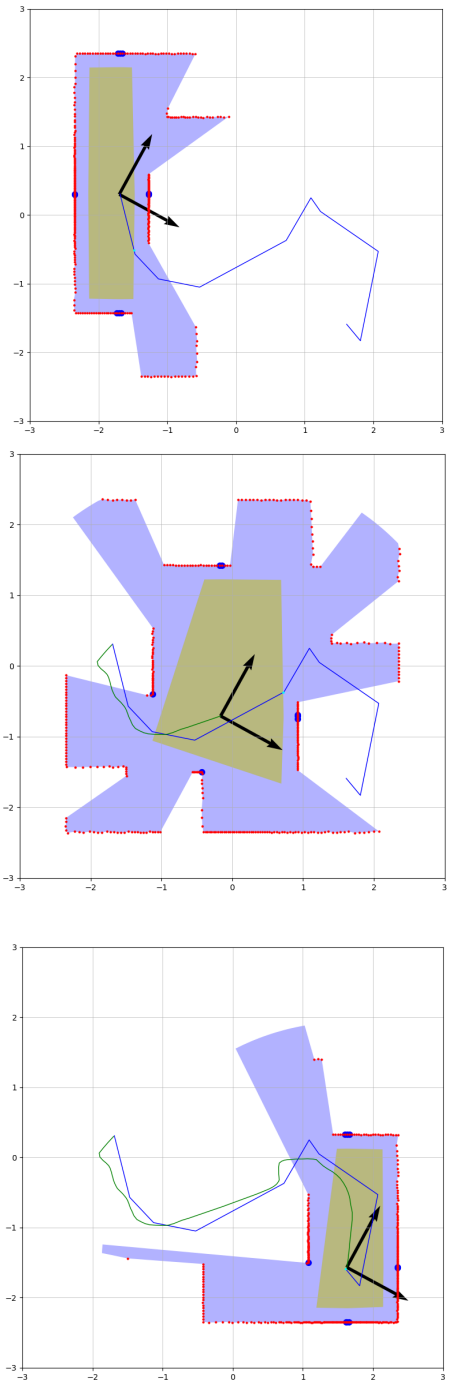

Fig. 7: The movement of the robot from its initial position to the goal, following the planned path

IV. Sampling- vs Search-based Mobile Navigation

A. Introduction

Compare the sampling-based mobile navigation solution with the previous search-based mobile navigation solution from Assignment 2, and comment on the pros and cons.

B. Comparison

In terms of computational efficiency, the search-based solution is expensive, when planning the path. The sampling-based solution is more computationally efficient as it reduces the number of nodes in the graph by constructing a sparse graph. Regarding optimality, Dijkstra over the whole costmap guarantees an optimal path but may be slow, while RRT* aims to optimize the path through sampling and graph refinement, offering a compromise between computational cost and optimality.

In terms of flexibility, RRT* is better suited to dynamic changes or new goals, as it can refine the existing graph rather than recomputing everything from scratch. The search-based approach, on the other hand, recalculates the path over the whole costmap for every change. Both approaches handle obstacles using a costmap, but RRT* also uses the concept of Local Free Space during graph construction.

In terms of implementability, RRT* is more complex to implement due to its reliance on techniques like weighted sampling and Bresenham's algorithm for collision checking.

C. Pros and Cons

The search-based solution guarantees an optimal path and has a relatively simple implementation, but it is computationally expensive and slow in complex environments, so it is less adaptable to changes. The sampling-based solution is more computationally efficient, adaptable to environmental changes, and generates a sparse graph, but it does not guarantee absolute path optimality and requires a more complex implementation.

D. Conclusion

The choice between the two solutions depends on the specific application requirements. If path optimality is critical and the environment is static, the search-based solution may be preferable. However, if computational efficiency and adaptability are priorities, the sampling-based solution is a better choice.

References

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