EE4308 Autonomous Robot Systems

Project 1

Philippe Brigger, Friedrich Ginnold
Maria Krinner, Philip Wiese

Contents

1 Introduction

This report is part of the module EE4308 Autonomous Robot Systems at the National University of Singapore (NUS) in spring 2022.
In summary, the following major modifications of the provided code have been made:

1. Path planning using Theta* (see Chapter 2)
2. Backwards path planning to find valid path when the goal point lies in an inaccessible area due to drifting (see Chapter 2)
3. Bidirectional motion controller (see Chapter 3)
4. Cubic Hermite spline trajectories (see Chapter 4)

Further, additional minor changes to support the before mentioned features were made and additional code for evaluation purposes was written.

- Publish current robot velocity to `vel_rbt` topic, as needed for cubic Hermite spline trajectory
- Automatic data collection for performance evaluation (see Section 5.2)
- Bugfixes of the original code (see Appendix A.1)
- Prevent flipping back occupied cells (see Appendix A.2)

Finally, the end results and evaluation is provided in Chapter 5.

2 Path Planning

2.1 Theta* Algorithm

The code which was given at the beginning of the project used a Dijkstra path planning algorithm. To optimize the path planning, the algorithm was adapted to Theta* [1, 2, 3]. The main difference between Dijkstra and Theta* is that Theta* checks whether there is a line of sight (LOS) between the parent and neighbor node of the current node. If this is the case, Theta* plans a path that goes directly from the parent to the neighbor node (see the green line in the scenario on the left of Figure 1). On the other hand, Dijkstra always plans a path that goes through the current node (see the blue line in the scenario on the left of Figure 1). Thus, the main advantage of Theta* is that it uses the most direct route to the neighbour node (which often times implies skipping the current node). If there is no LOS, then Theta* applies the Dijkstra path planning algorithm as seen in the scenario on the right of Figure 1.

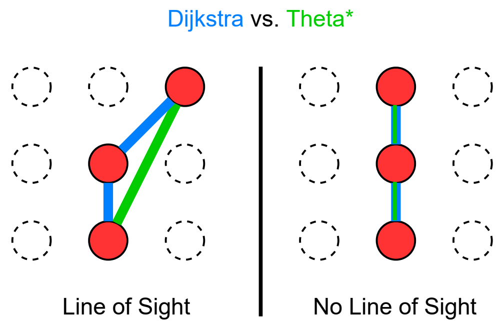


Figure 1: Theta* checks whether the neighbor node is in the line of sight of the parent node. If this is the case, then Theta* does not plan a path through the current node. On the other hand, Dijkstra always goes through all three nodes.

Using a LOS approach for path planning makes Theta* an optimal any-angle planner. Initially, a custom LOS function `getLine()` was written to check whether a neighboring node was in the LOS of the parent node. This was done by defining a linear function $f(x) = mx + b$ between the parent and the neighbor and asserting whether the current node lies within f(x). However, the custom function did not run correctly, which is why the algorithm was adapted to replicate the pseudo-code given in the lecture slides (02_Planners, 41-43). One of the problems with the original LOS formulation was that it did not assert whether the intermediate nodes of the LOS were located in the map and were not inflated. The final algorithm can be synthesized into the following steps:

1. Check whether neighbor lies in map and is not inflated
2. Check whether neighbor lies in sight of parent
3. Add euclidean distance from parent to neighbor node to g cost of neighbor node
4. Update g cost of neighbor node if it is cheaper than the current g cost at neighbor node
5. Append neighbor node to node list
6. If goal is reached, retrieve path indices from start to goal index

2.2 Backwards Path Planning

If the goal lies in an inflated region, the planner cannot plan a valid path and the robot gets stuck. To prevent this from happening, the path planning algorithm first checks if the goal is inflated. If it is not inflated, the path is planned with Theta* from the start node, n_0, to the goal node, n_g. However, if the goal is inflated, backwards planning is done from n_g to n_0 instead of from n_0 to n_g. This is done by initially swapping the start and goal indices. Then, the same Theta* path planning algorithm as described in Section 2.1 is used. When the path planning is finished, the path will contain the nodes in an order that describes the path from the n_g to the n_0. Since the robot must drive from n_0 to n_g in practice, the order of the indices in the path has to be reversed. Additionally, the parent attribute of each node in the list must be updated according to the new index order.
To conclude, if n_g is inflated, the steps that the path planner will take are summarized as follows.

1. Check whether goal is inflated
2. If goal is inflated, swap start and goal indices
3. Do normal Theta* path planning as described in Section 2.1
4. Reverse node list containing path
5. Adapt node parents such that path runs from start to goal

2.3 Simulation of Theta*

The total g cost of each goal is used as a metric to benchmark the performance of Theta* against Dijkstra. For each goal, every time the path is re-planned, the maximum g cost of the current path is compared against the maximum g cost of the previous paths and the largest is saved. This provides a worst case scenario for the path planner, which is used as the main criteria for comparison. The process is averaged over ten different simulation runs with a grid cell size of 0.05. The average g costs to reach the three goals are listed in Table 1. The values showed that the

Table 1: Average g cost to reach goal 1, 2 and 3 for Theta* and Dijkstra path planning algorithm. Theta* has cheaper average g costs to reach the goals.

Path Planner	Goal 1	Goal 2	Goal 3
Theta*	80.6	121.6	137.6
Dijkstra	84.1	129.0	137.7
Relative cost	95.8 %	94.3 %	99.9 %

paths planned by the Theta* algorithm had lower g costs than the paths planned by the Dijkstra path planning algorithm. Therefore, the data justified the choice of extending the Dijkstra to a Theta* path planner.

3 Bi-Directional Motion Controller

3.1 Linear Motion Controller

To make the robot move forwards or backwards towards a target, the controller needs to check if the target (x_p, y_p) is generally in front of or behind the robot (x_k, y_k). This can be examined by considering the angular error $\epsilon_{k,\theta}$, which is defined as

$$\epsilon_{k,\theta} = atan2(y_p - y_k, x_p - x_k) - \psi_k \ , \quad -\pi \leq \epsilon_{k,\theta} < \pi \tag{1}$$

The general motion controller calculates the linear velocity control input $u_{k,r}$ by coupling the PID controlled velocity with the angular error as

$$u_{k,r} = (P_{k,r} + I_{k,r} + D_{k,r}) \cdot g(\epsilon_{k,\theta}) \tag{2}$$

For the uni-directional controller from Lab 2, $g = g_{uni}$ was chosen. Note that $g_{uni}(\alpha)$ is a continuous function that outputs a value between 0 and 1 for $-\pi \leq \alpha < \pi$, with $g_{uni}(0) = 1$ and $g_{uni}(\{|\alpha| \geq \beta\}) = 0$ for some $0 < \beta \leq \pi$.
The bi-directional motion controller takes into account that the robot drives backwards when the target is located behind the robot and forwards when the target is located in front of the robot. The robot's direction comes from the g function, because $(P_{k,r} + I_{k,r} + D_{k,r}) > 0$. A continuous function $g = g_{bi}$ must be chosen such that $g_{bi}(\alpha) \in [-1, 1], \forall \alpha \in [-\pi, \pi)$. To take the direction into account, $g_{bi}(\epsilon_{k,\theta})$ must be designed as follows:

$$g_{bi}(\epsilon_{k,\theta}) > 0 \ , \quad \forall \ |\epsilon_{k,\theta}| < \pi/2 \ , \quad \text{to drive forwards} \tag{3}$$

$$g_{bi}(\epsilon_{k,\theta}) < 0 \ , \quad \forall \ |\epsilon_{k,\theta}| > \pi/2 \ , \quad \text{to drive backwards} \tag{4}$$

This leads to $g_{bi}(-\pi/2) = g_{bi}(\pi/2) = 0$.
For a good coupling between linear velocity and angular error $g_{bi}(0) = 1$ and $g_{bi}(-\pi) = g_{bi}(\pi) = -1$ are desired. A suitable function would be $g_{bi} = cos^3$ which leads to

$$u_{k,r,bi} = (P_{k,r} + I_{k,r} + D_{k,r}) \cdot cos^3(\epsilon_{k,\theta}) \tag{5}$$

3.2 Angular Motion Controller

The angular error $\epsilon_{k,\theta}$ is always calculated against the robot's forwards directed axis. To implement a bi-directional motion controller, the angular error to the axis in which the robot is moving must be considered. The directed angular error $\bar{\epsilon}_{k,\theta}$ that is calculated against the robot's forward directed axis during forward movement and against the robot's backwards directed axis during backwards movement is introduced. This leads to the following equation:

$$\bar{\epsilon}_{k,\theta} = \begin{cases} \epsilon_{k,\theta} - \pi, & \text{if } \epsilon_{k,\theta} > \pi/2 \\ \epsilon_{k,\theta} + \pi, & \text{if } \epsilon_{k,\theta} < -\pi/2 \\ \epsilon_{k,\theta}, & \text{otherwise} \end{cases} \tag{6}$$

The PID controller for the angular motion from Lab 2 needs to consider $\bar{\epsilon}_{k,\theta}$.

4 Spline Trajectory Generation

Due to the trade-off between computational complexity and smoothness, cubic Hermite splines were used for trajectory fitting. Cubic Hermite splines ensure C^1 continuity. Hence, the points are joined with the same gradient and, consequently, with the same velocity.
Following the lecture notes, the trajectory can be calculated with

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{7}$$
$$\dot{x}(t) = a_1 + 2a_2 t + 3a_3 \tag{8}$$
$$y(t) = b_0 + b_1 t + b_2 t^2 + b_3 t^3 \tag{9}$$
$$\dot{y}(t) = b_1 + 2b_2 t + 3b_3 \tag{10}$$

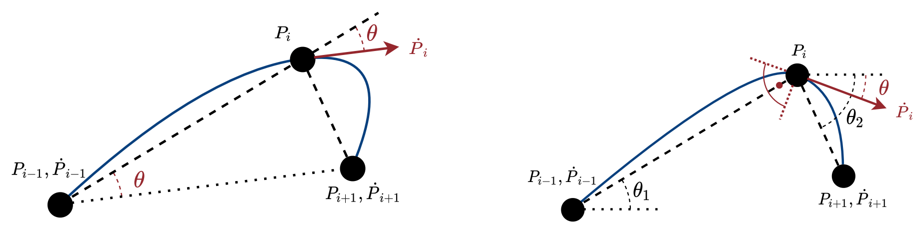
The parameters of the cubic Hermite spline are calculated with

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \end{bmatrix}^\intercal = M^{-1} \begin{bmatrix} x_{i-1} & \dot{x}_{i-1} & x_i & \dot{x}_i \end{bmatrix}^\intercal \tag{11}$$
$$\begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix}^\intercal = M^{-1} \begin{bmatrix} y_{i-1} & \dot{y}_{i-1} & y_i & \dot{y}_i \end{bmatrix}^\intercal \tag{12}$$
$$\tag{13}$$

with M^{-1} defined by

$$M^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{\Delta t^2} & -\frac{2}{\Delta t} & \frac{3}{\Delta t^2} & -\frac{1}{\Delta t} \\ \frac{2}{\Delta t^3} & \frac{1}{\Delta t^2} & -\frac{2}{\Delta t^3} & \frac{1}{\Delta t^2} \end{bmatrix} \tag{14}$$

In order to calculate the velocity vector at the turning point, two approaches were investigated, as shown in Figure 2.
For both methods, the magnitude of the velocity vector at each point (except for the starting point) was set to the desired average speed v_{avg} of the robot. For the starting point of the trajectory, the magnitude of the velocity vector was set as the currently estimated robot velocity v_t since the robot already was at this location.



(a) Intermediate Angle (b) Average Angle

Figure 2: Angle calculation of velocity vector for spline fitting.

The simpler method is to set the direction of the velocity vector at the point P_i equal to the heading between the point P_{i-1} and P_{i+1} as shown in Figure 2a.

$$\theta = \text{atan2}(P_{i+1,y} - P_{i-1,y}, P_{i+1,x} - P_{i-1,x}) \tag{15}$$

However, this approach leads to suboptimal results as the angular accelerations on the trajectory are strongly non-homogeneous. A better approach is to average the

directions between the points. Consequently, the direction of the velocity vector is perpendicular to the bisector of the two partial paths as shown in Figure 2b.
The heading between the points can be calculated with:

$$\theta_1 = \text{atan2}(P_{i,y} - P_{i-1,y}, P_{i,x} - P_{i-1,x}) \tag{16}$$

$$\theta_2 = \text{atan2}(P_{i+1,y} - P_{i,y}, P_{i+1,x} - P_{i,x}) \tag{17}$$

$$\theta = \begin{cases} \frac{1}{2}(\theta_1 + \theta_2), & \text{if } |\theta_1 - \theta_2| \leq \pi \\ \frac{1}{2}(\theta_1 + \theta_2) - \pi, & \text{otherwise} \end{cases} \tag{18}$$

with θ_1, θ_2 converted to the range $[0, 2\pi]$. Explained in simple words, the robot will already have performed half of the required rotation when reaching the point P_i. Hence, the velocity vector at the point P_i can be calculated with:

$$\dot{P}_{i,x} = cos(\theta) * ||\dot{P}_i|| \tag{19}$$

$$\dot{P}_{i,y} = sin(\theta) * ||\dot{P}_i|| \tag{20}$$

4.1 Simulation of Spline Trajectory

The correct functionality of the spline trajectory was successfully tested in simulation and the result is shown in Figure 3.
The blue line indicates the path provided by the global planner after pre-processing. The red line represents the corresponding cubic Hermite spline trajectory.

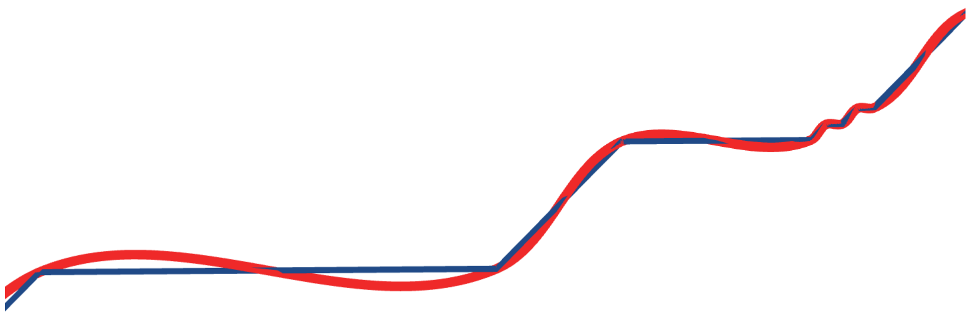


Figure 3: Cubic Hermite spline trajectory with the averaged angle approach.

5 Results & Evaluation

5.1 Problems

During the evaluation on the real hardware several problems were encountered.

- **Large angular drifting:**
  To deal with the angular drifting, the motion filter parameters were adjusted until the robot successfully drove to all three goals within a circle of radius $r = 30cm$. The chosen parameters are `weight_odom_v=0.75`, `weight_odom_w=0`.

- **Invalid paths if goal lies in inaccessible area:**
  Initially, only forward planning was implemented, which led to the robot moving towards the dead end direction and eventually getting stuck. This is circumvented by implementing the backwards planner, as explained in Section 2.2. In this particular scenario, it means that the robot detects that the third target lies in an inaccessible area. If the robot instead starts planning from the goal towards the start (backwards), it will no longer move towards the dead end and thus choose the correct lane.

- **Wrong direction of velocity for spline trajectory:**
  In the first implementation of the spline trajectory, the direction of the velocity vector sometimes was facing the wrong way. The reason for this is that the information about the direction is lost if the mean value of the angles is naively calculated. Hence, a case distinction as presented in Equation 18 has to be used.

5.2 Automatic Data Collection

To make the data evaluation and results logging processes as efficient as possible, several steps were automated. The data of each test run with the real robot was saved in a .bag file. The .bag file was then imported into a script to plot and analyze the results. Furthermore, publishers were used to e.g. share the maximum g-cost of the planned paths.

5.3 PID Controller Tuning

Suitable PID gains for the linear and angular velocity control inputs were found using the automatic data collection and evaluation. Several simulations were run in the provided gazebo world `World21` with different controller gains and the corresponding angular and linear errors were compared. The linear and angular error plots of two such runs can be observed in Figure 4. The left plot (a) shows the errors for gains $K_{P,lin} = 1.0, K_{D,lin} = 0.01, K_{P,ang} = 5.0, K_{D,ang} = 0.01$ while the right plot (b) had higher PID gains $K_{P,lin} = 10.0, K_{D,lin} = 0.1, K_{P,ang} = 50.0, K_{D,ang} = 0.1$. Whereas the linear error in (a) is mostly around 0.1m with few spikes, there are multiple spikes of up to 0.2m in the linear error for (b). A suitable configuration should lead to a constant but small error while following a target and few spikes should occur during re-planning. The angular error for (a) has only one spike during the re-planning for the second goal and oscillates around zero with an amplitude of at most 0.5 rad. For configuration (b) a similar amplitude but more spikes than for (a) can be observed. A suitable configuration should give few spikes and a small error amplitude. The high gains in (b) lead to faster runs, as the robot drives in saturation most of the time. Nevertheless, the movement of the robot seems less controlled, less elegant and the path has to be re-planned often. This is why configuration (a) was preferred for the real world tests. Configurations where $K_{I,lin} \neq 0$

or $K_{I,ang} \neq 0$ were not considered, as the robot only had to be within a circle with radius $r = 0.3\,\mathrm{m}$ of the goal.

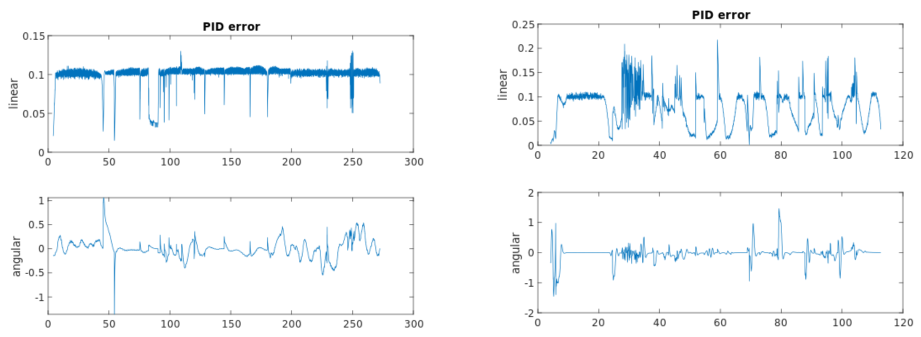


Figure 4: Linear and angular error in m and rad between robot and target for different PID gains.

5.4 Real-World Experiment

Several real-world experiments were conducted in the laboratory. Three different run configurations are presented in this section. In all cases, the planner is able to switch between the forward and backward configurations as explained in Section 2 and the robot visits all three goal targets. Each of the figures depicts the estimated (`/turtle/pose`) and target (`/turtle/target`) trajectories over the log odds map (`/turtle/grid/log_odds`). In the first run configuration the Dijkstra algorithm was used (see Figure 5). Since there is no possibility to skip nodes in order to plan a more direct route, the trajectory has to be re-planned more frequently. This is observed from the drastic changes in the target trajectory.

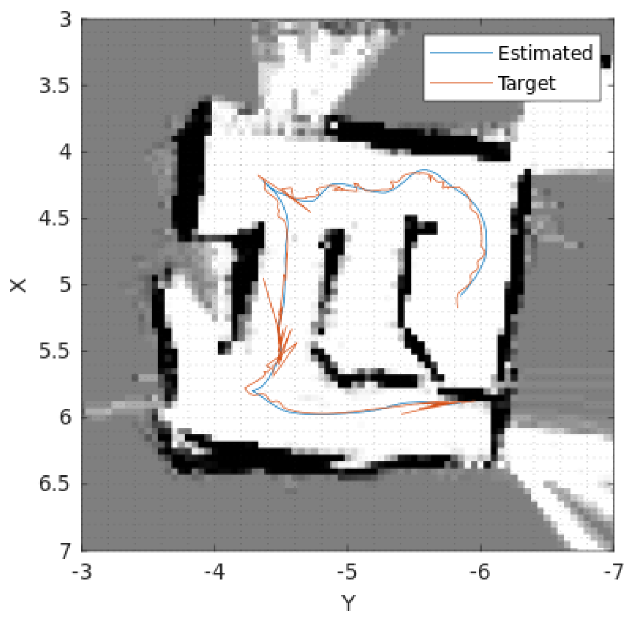


Figure 5: Estimated and target position over occupancy grid using Dijkstra planner.

A smoother trajectory with less re-planning is accomplished using the Theta* algorithm. As shown in Figure 6, the target trajectory is planned less frequently due to the direct planning when there is line of sight. When approaching the third target, the robot first moves towards the dead end before it realizes that the goal is on the other side of the obstacle barrier. Since the third goal is inflated, the robot switches to backwards planning, which allows to quickly identify that it needs to approach the target from the other side of the obstacle barrier. Note that this was not possible with the original forward planning. Furthermore, it was observed that the behavior is not deterministic and depends on each individual run attempt.

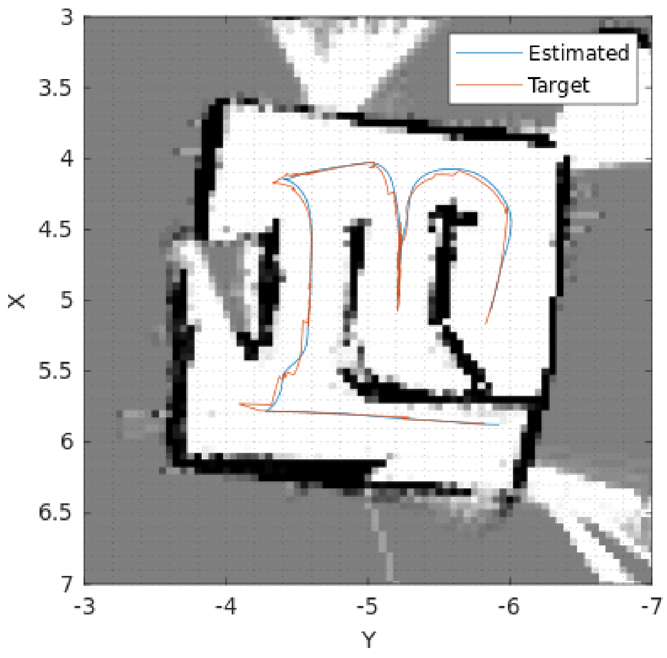


Figure 6: Estimated and target position over occupancy grid using Theta* planner.

Finally, in order to gain a better understanding of the effect of splines, a third run was carried out without splines. The remaining configuration parameters were chosen as for the second run. As can be observed in Figure 7, the estimated trajectory has sharper edges at $x = 4.2, y = -4.9$ compared to Figure 6. Moreover, the curve around the second obstacle barrier is less smooth when removing the splines. On the other hand, it was observed that removing the splines led to a significant reduction of the computational time.

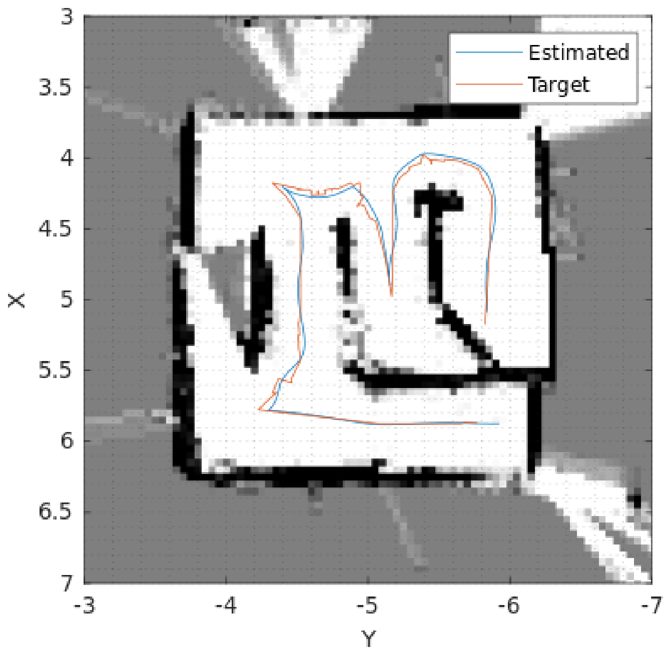


Figure 7: Estimated and target position over occupancy grid without using splines.

References

[1] Daniel, K., Nash, A., Koenig, S., and Felner, A. (2010). Theta*: Any-angle path planning on grids.

[2] Nash, A., Tovey, C. A., Koenig, S., and Tovey, C. (2010). Lazy theta*: Any-angle path planning and path length analysis in 3d. lazy theta*: Any-angle path planning and path length analysis in 3d.

[3] Nawi, I. M. and Rahim, M. A. S. A. (2008). Autonomous mobile robot: Path planning using backward chaining. pages 182–191.

A Bugfixes

Several bugs were discovered during the project. The bugs and the corresponding solutions are explained in this chapter.

A.1 Negative Index

Note that this error was discovered before the official announcement on LumiNUS. The pre-increment operation on line 8 of Listing 1 is wrong and leads to unexpected behavior. The reason for this is that, together with the other pre-increment operation on line 14, the value of t can actually get lower than one.

Listing 1: Partial main.cpp

```cpp
 1 // [...] More code
 2 
 3 // Always try to publish the next target so it does
 4 // not get stuck waiting for a new path.
 5 if (dist_euc(pos_rbt, pos_target) < close_enough)
 6 {
 7     // The --t statement is not correct and
 8     // leads to invalid pos_targets.
 9     if (--t < 0)
10         t = 0;
11 
12     pos_target = trajectory[--t];
13     ROS_INFO(" TMAIN : Get next target (%f,%f)",
14         pos_target.x, pos_target.y);
15 
16     // publish to target topic
17     msg_target.point.x = pos_target.x;
18     msg_target.point.y = pos_target.y;
19     pub_target.publish(msg_target);
20 }
21 
22 // [...] More code
```

Hence, statement on line 9 in Listing 1

```cpp
if (--t < 0)
```

has to be changed to

```cpp
if (t < 0)
```

in order to ensure correct behavior.

A.2 Flipping Back Occupied Cells

During the real-world experiment, it was observed that already occupied cells can be flipped back to a free state. This was later also verified with simulations. This can be quite problematic when, due to drift between the estimated and the real position of the robot, new measurements lead to inconsistency in the occupancy grid. This can result in changing occupied cells into unoccupied ones and may cause holes in previously detected walls. Hence, it happens that the robot explores a new path, detects a dead end, moves back and then "forgets" that the previously explored area is a dead end.
One feasible approach is to prevent flipping occupied cells back to an occupied state.

Listing 2: Partial grid.cpp

```cpp
1  // [...] More code
2  
3  void Grid::change_log_odds(bool occupy, Index idx) {
4      if (out_of_map(idx))
5          return; // ignore points that are out of map
6  
7      // 1 if observed to be occupied, -1 otherwise
8      int inc = occupy ? 1 : -1;
9  
10     int k = get_key(idx);
11     // copy of previous log odds value
12     int prev_log_odds = grid_log_odds[k];
13 
14     if (occupy && prev_log_odds < log_odds_cap) {
15     // Add 1 if less than cap and cell
16     // is observed to be occupied
17         ++grid_log_odds[k];
18     } else if (!occupy && prev_log_odds > -log_odds_cap) {
19     // Subtract 1 if more than cap and cell
20     // is observed to be free
21         --grid_log_odds[k];
22     }
23     if (prev_log_odds < log_odds_thresh
24         && grid_log_odds[k] >= log_odds_thresh) {
25     // unknown log odds becomes occupied log odds
26 
27         // add inflation due to new occupied
28         // status at cell (i, j)
29         change_inflation(true, idx);
30     } else if (prev_log_odds >= log_odds_thresh
31         && grid_log_odds[k] < log_odds_thresh) {
32     // occupied log odds becomes unknown log odds
33 
34         // remove inflation from previous
35         // occupied status at cell (i, j)
36         change_inflation(false, idx);
37     }
38 }
39 
40 // [...] More code
```

For this, the condition in line 18 of Listing 2

```cpp
else if (!occupy && prev_log_odds > -log_odds_cap)
```

was changed to

```cpp
else if (!occupy && prev_log_odds > -log_odds_cap
  && prev_log_odds < log_odds_thresh)
```

Note that this leads to an overestimation of the number and position of the occupied cells, but leads to more stable behavior in the presence of drift.