INFO0948-1: Midterm report

Antoine Debor (s173215) Pierre Navez (s154062)

I. Map

In order to represent the map graphically, a 150×150 grid has been initialized. Knowing that the robot is in a square room of 15 meters by 15 meters, it means that one cell of the grid represents a 10 by 10 centimeters square of the room. This resolution has been chosen for two reasons. First, the smallest dimension the robot can find is the wall's width inside the room and it is about 11 centimeters. Secondly, more cells in the grid would mean longer computation times to fill in the grid, and too much precision is not necessary because the map will eventually be used to avoid obstacles that are supposed to be bigger than 10 centimeters. Moreover, this grid will be used for the path planning and if it is too big, it will slow down the algorithm. The strategy adopted to draw the map inside the grid was to consider three states, represented by 0s, 1s and 2s. Initially each cell contains a 1, meaning that it is unexplored, in other words, not sensed by the Hokuyo sensor yet. This state turns into 0 when the cell is explored and free, or into a 2 when it is explored and identified as being part of an obstacle. Note that, in order to speed up the computation, the Hokuyo sensor's data has been downsampled by a factor 10, which still leads to satisfactory results.

II. Path planning

After some research on path planning algorithms, it was first planned to use "D* lite" [1] which is an algorithm that combines incremental and heuristic search. The paper presenting it shows that even if it sub-optimal, it is fast and robust. However, we could not find any simple implementation or ready-to-use toolbox. It was then chosen to use DXform (Distance transform navigation), coming from Peter Corke's book (section 5.2.1) [2] and implemented inside his robotics toolbox for *Python* [1], which is quite simple to use and which is quite fast in the considered environment. Basically the algorithm works as follows: a grid is filled with 0s and a 1 at the goal position. Then, the distance from every point to the goal is computed using a binarized version of the current map and filled in this matrix. These distance actually take into account the presence of observed obstacles in the field. The path that the robot will follow is simply the sequence of shortest distance from its position to the neighboring cells. The goal is reached when the smallest distance is zero. Note that the binary map corresponds to an inflated version of the current map, *i.e.* the obstacles have been inflated to avoid collision.

[1]The toolbox can be found at `https://github.com/petercorke/robotics-toolbox-python`.

The complexity of the algorithm is $\mathcal{O}(N^2)$, where N is the largest dimension of the grid. This justifies why the grid of the map has not been taken too large.

When the robot is at idle state, the frontier between the explored and unexplored space is computed and stored in a data structure. The target selection strategy is the following: after a first half turn, the robot selects at random a point on this boundary as the next target, but for the subsequent target selections, it chooses the one leading to the smallest sum of its Euclidean distance to the robot and its distance according to the distance map computed by DXform with the robot's position given as the goal, as it can be seen in `target_seeker.py`. The first random pick has been chosen assuming that there is no smarter way than another to select a first target, not knowing much about the environment. The strategy for the subsequent targets aims at providing a quite continuous path along the house, avoiding as much as possible unnecessary back and forth trips while taking into account the presence of walls (which would not be the case if only the Euclidean distance was considered).

It is also noteworthy that such a path planning algorithm will compute trajectories that are not smooth and it is sub-optimal for the robot to strictly follow this trajectory point by point. We first tried to downsample and/or interpolate the trajectory using `scipy.interpolate.splprep` but, since it did not yield satisfactory results, we implemented a more clever trajectory post-processing function in `trajectory_smoother.py`. This function only keeps critical and relevant steps in a trajectory (sharp turns, goal, ...) rather than the whole set of points.

III. Controller

It has been decided to use a proportional controller to drive the robot. For the forward velocity control, the velocity is proportional to the Euclidean distance from the robot to the next trajectory step, in order to not miss any step as this could lead to unwanted detrimental behaviours in critical areas. For the orientation of the robot, however, the rotation speed is set to zero until the angle between the robot's Hokuyo sensor side and the next step becomes too large. At this moment, the forward velocity is momentarily set to zero and the rotation velocity is proportional to the difference in orientation. Once this difference is small enough, the rotation speed is set back to zero and the forward velocity becomes proportional to the distance again. In other words, the strategy/policy could be summarized as "rotate only when it is really required, but do it precisely".

IV. Finite state machine

Figure 1 represents the finite state machine of the robot. A transition from one state to another either occurs when the current process associated to the state finishes, or when a particular condition is encountered, in which case this condition is written synthetically above the specific arrow corresponding to this transition.

V. Computation time

Figure 2 presents the histogram of the computation time of the loops for one instance of the exploration phase. It can be seen that most of the loops are below 50 ms, even if the maximum one (*i.e.* the blue dashed line) is way larger. These larger times correspond to the trajectory post-processing procedures but, since during these the robot is idle, it does not lead to any detrimental effect.

VI. Video

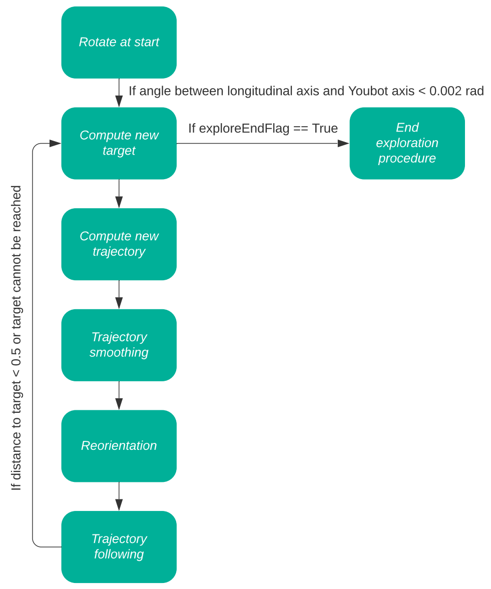
A commented video of the exploration phase can be seen *here*.

References

[1] M. L. Sven Koenig, “D* lite,” *Proceedings of the AAAI Conference of Artificial Intelligence (AAAI)*, pp. 476–483, 2002. [Online]. Available: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf

[2] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 2nd ed. Cham: Springer, 2017.

Appendix



Rotate at start
If angle between longitudinal axis and Youbot axis < 0.002 rad
Compute new target
If exploreEndFlag == True
End exploration procedure
Compute new trajectory
Trajectory smoothing
Reorientation
Trajectory following
If distance to target < 0.5 or target cannot be reached

Fig. 1. Finite state machine of the robot’s behaviour

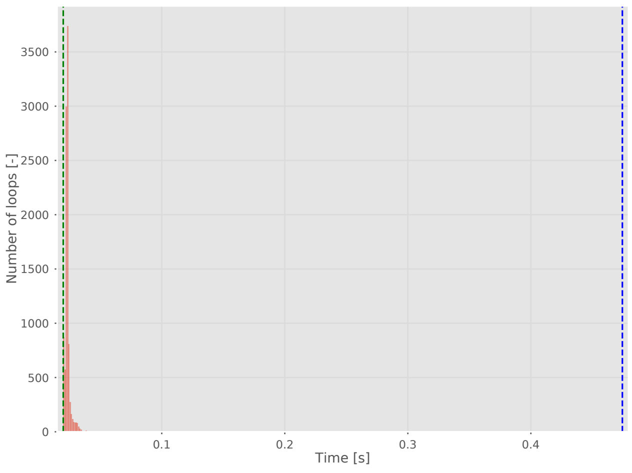


Fig. 2. Histogram of the loops’ computing time