INFO0948-1: Final report

Giulio Corsi (s208018)   Giacomo Velo (s208001)

I. Introduction

In this final report, the development and results of the Navigation and Manipulation milestones will be presented.
For the Navigation part, the first two milestones (1.i) and (1.ii) were completed. In (1.i), the youBot is exploring the map and realizing a complete representation of it accessing the GPS info (`simxGetObjectPosition`) and using `simxGetObjectOrientation` once in every cycle. In (1.ii), the GPS info were taken only one time at the beginning of the computation, while the orientation was taken again once in every cycle. In both cases, the youBot is able to complete the navigation and to produce a map of the surrounding environment, even if in the second case the map results to be more inaccurate.
For the Manipulation part, the milestone (2.i) was completed: the youBot is able to grab all the objects on the first table (where objects stand upright) and put them on the target table, without any falling on the ground.

II. Navigation using GPS coordinates

For this milestone, the youBot will need to navigate in an unknown environment and map it. The main state machine is presented in Fig. 1 and is used also for the milestone (1.ii).

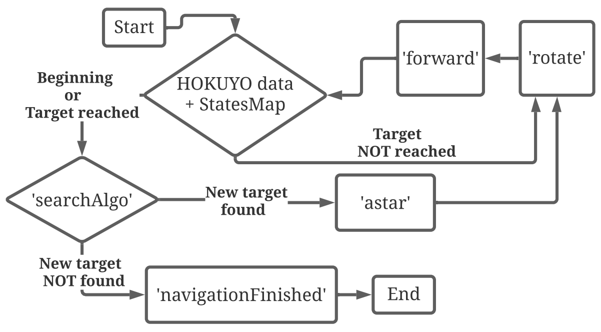

Fig. 1. State machine diagram.

The map of the room is built up creating a matrix (StatesMap) and associating a specific number to each of its elements. The dimension of the matrix is dependent on the resolution selected. The values assigned to the elements are: 0: free point, 1: unknown point, 2: obstacle, 3: point close to an obstacle. The following steps explain which is the logic behind the exploration:

1) `HOKUYO data` **and** `StatesMap`: the position of the youBot is obtained from the GPS. The outputs of the sensor are collected and organized in the statesMap. The obstacles are assigned with state 2 and their neighboring points with state 3. A copy of the statesMap is realized and the value 4 is assigned to the neighboring points of state 3, consequently, the value 5 is assigned to the neighboring points of value 4.
2) `SearchAlgo`: the target is searched. The first attempt consists of analyzing the points close to the horizon of the sensor. Comparing the latter with the StateMap it is possible to understand if there is an unknown point close to the horizon. If yes, the free space adjacent to the latter is selected as the target. If no, the entire StateMap is analyzed and the nearest unknown point next to a free space is selected as the target. This to let the youBot start the exploration from the closest points to itself and so have an efficient map investigation. A minimum distance is selected to let the youBot look for a path longer than a threshold among the available points. When it is not possible to find an unknown point adjacent to a free space the state machine enters *NavigationFinished*. The remaining unknown spaces are unreachable and are so assigned with state 2.
3) `astar`: creation of a path between the position of the youBot and the target found. To do so, an implementation of the A* algorithm by Melvin Petrocchi [1] is used. It has been chosen this shortest path algorithm instead of dynamic programming because, after founding the target, there are only one origin and one destination: in this case, most of the nodes are not relevant and A* can be more efficient than a DP algorithm where every node participate in the computation.

Also, a Probabilistic Roadmap method could be used, but this does not guarantee that all areas of interest are explored, which is the main focus of this task. The A* algorithm is a modification of Dijkstra's algorithm that implements a heuristic function to speed up the search [2]. The D* algorithm also adds the incremental replanning with respect to A*, but no implementation has been found for Python. A* can be fed with a weighting matrix: a decreasing weight has been assigned to the statesMap points which have values of 2, 3, 4, and 5 respectively. In this way, points far at least four grid boxes to the obstacles are privileged in the construction of the path.

4) `rotate`: the youBot rotates in the direction of the next point of the path. The angular velocity is proportional to the angular distance between the youBot angle and the final angle and stops when this difference is smaller than a tolerance.
5) `forward`: the youBot goes forward in order to reach the next point of the path. The velocity is constant: it was seen that this speeds up the navigation and does not cause particular issues, at least at this stage. After this state, the cycle restart from updating the StatesMap.

Results

Here, the final results are presented. In Fig. 2 the final map is shown. As it can be seen, all the environment is mapped without errors.

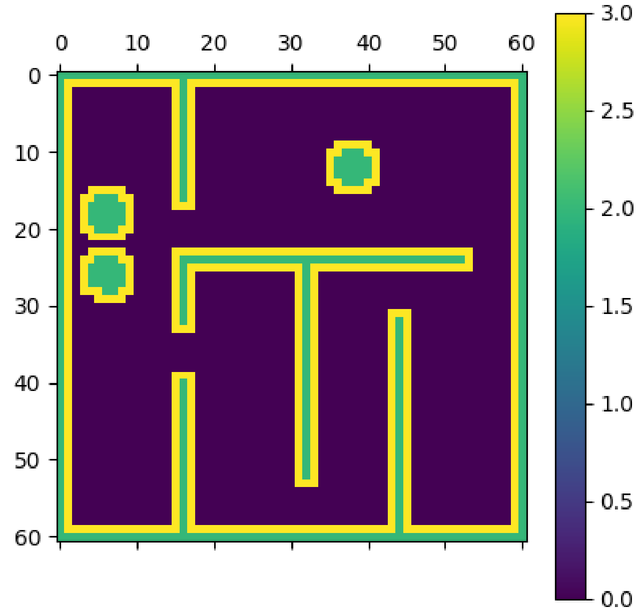

Fig. 2. Final map.

The yellow points of the grid represent free cells that are near to the obstacles. The resolution of a single cell was set to 0.25 as a tradeoff between accuracy and computational time, but it can be changed to other values.
In Fig. 3 the minimum and maximum times without and with the SynchronousTrigger and the times histograms for a complete run are depicted.

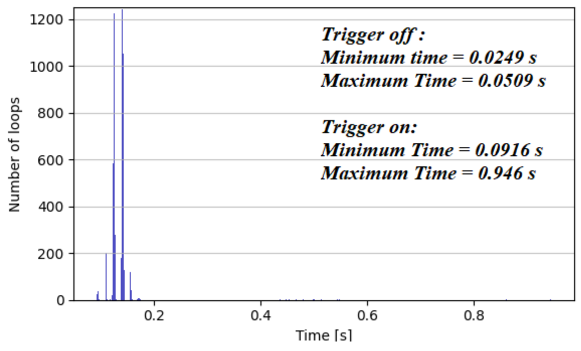

Fig. 3. Times histograms for a complete run.

III. Navigation using distance from beacons

For this second milestone, as already stated, the state machine diagram is the same of Fig. 1. The difference now is in the *HOKUYO data and StatesMap* state: instead of using `simxGetObjectPosition`, the youBot has access to its distance from three beacons sending radio signals through the sensor. This information can be obtained through the function `youbot_beacon` with `noise=True`, which returns a noisy output from the sensor. The distance from the beacons is used to evaluate the youBot position in the following way: at the beginning, the position of the beacons is obtained through `simxGetObjectPosition`. With this information, it is possible to draw three circumferences that have as center the position of the beacons and as radius the distance obtained previously. Now, if there had been no noise, these three circumferences would have had a single common intersection, which would have represented the position of the youBot. The noise causes there to be three points of intersection between the circumferences: once these three points have been obtained, the midpoint between them has been calculated and assumed as the position of the youBot. Comparing the position evaluated in this way with the real position obtained from `simxGetObjectPosition`, it can be seen that the youBot estimate its position quite well, but still too inaccurately, because of the uncertainty added by the noise. For this reason, it was decided to implement also a particle filter. This is a recursive, Bayesian state estimator that uses discrete particles to approximate the posterior distribution of an estimated state. To do so, it was used an implementation of the particle filter by John H. Williamson [3]. To setup the filter, it is required to specify an *observation function*, an *initial distribution* and a *weight function*. In addition to these, also a *dynamics function* was added (forward prediction), with *diffusion* (to add noise).

A. *Initial distribution*

As initial distribution, it was decided to use a normal distribution of particles on the whole map. At the beginning, the normal distribution was centered in the origin with a standard deviation related to the map size. Different trials were made with a different number of particles (100, 500, 1000). In all the cases, the results were inaccurate and a large number of the particles were irrelevant to the computation because too far from the youBot position. For this reason, it was decided to center the distribution of the particles directly in the youBot position already calculated before, so that every particle could be useful. Moreover, it was selected a resample proportion of 0.5 to try to improve the model.

B. *Weight function*

The weight function chosen was a squared error function, which compares the hypotethical observed arrays coming from the particles with the real observation provided by the sensor, and calculates the squared error of each particle in order to assign a positive weight value to each particle, that is then used to do the resampling and to estimate the final position.

C. *Dynamics function*

The *dynamics function* is used to update the state based on internal (forward prediction) dynamics. Therefore, it has to simulate the youBot movement in the best possible way in order to displace the particles correctly and to approximate at best the youBot position. In the *dynamics function*, the coordinates of the particles $[x_{\text{part}};\ y_{\text{part}}]$ are updated by adding to their previous coordinate $[x_{\text{part}};\ y_{\text{part}}]$ respectively the displacement made by the youBot in the x-axis, Δx and y-axis, Δy. Those displacements are computed by multiplying the speed at which the youBot is moving and the duration of this action. Knowing the orientation angle of the youBot, it is possible to determine the velocities in the x and y axis by using sine and cosine functions as:

$$x_{\text{part}} = x_{\text{prev}} + v \cdot \Delta t \cdot \sin(\alpha) \tag{1}$$

$$y_{\text{part}} = y_{\text{prev}} - v \cdot \Delta t \cdot \cos(\alpha) \tag{2}$$

where the velocity v is constant, the timestep Δt is equal to 0.05 seconds and α is the orientation of the youBot.
In addition to this, some noise was added to take into account the uncertainty in the movement. In *pfilter*, this can be added by specifying a *diffusion function*: in this case, it was selected the Cauchy probability distribution `cauchy_noise` with scale factor of 1 mm.

D. *Observation function*

The *observation function* is a function that returns a predicted observation for each particle and it is a fundamental step of the particle filter: this predicted value for each particle will be compared to the real observed state and the weights of the particles will be evaluated in relation to the error between the predicted value and the observed state. Since the position of the beacons is known, the *observation function* has been defined as the distance between the particle and the centers of the beacons obtained previously. This will be compared to the real observed state which is the distance taken from `youbot_beacon`.

Results

In the end, the youBot manage to navigate throughout the whole environment and build a map of it, shown in Fig. 4. As it can be seen, the map is not very accurate: in particular, the walls are not well represented and they are often thicker than the real ones, but anyway the walls, the tables, and the doors are all identified and the inaccuracies do not cause a premature ending of the navigation.
In order to improve the accuracy of the map, the setup of the particle filter should be analyzed more deeply. In fact, a trial of the navigation without the filter was also made (using as position the mean value of the intersections between the circumferences) and the final map was quite the same: this means that the particle filter does not change too much the estimation of the position. For sure, one way to refine the map is to increase the number of particles. The map in Fig. 4 was generated with 1000 particles and is better than the one generated with 100 particles, but the small advantage in terms of accuracy is outweighed by the

longer time required for the computations. A better solution could be refining the *dynamics function* and defining the *observation function* in a different way .

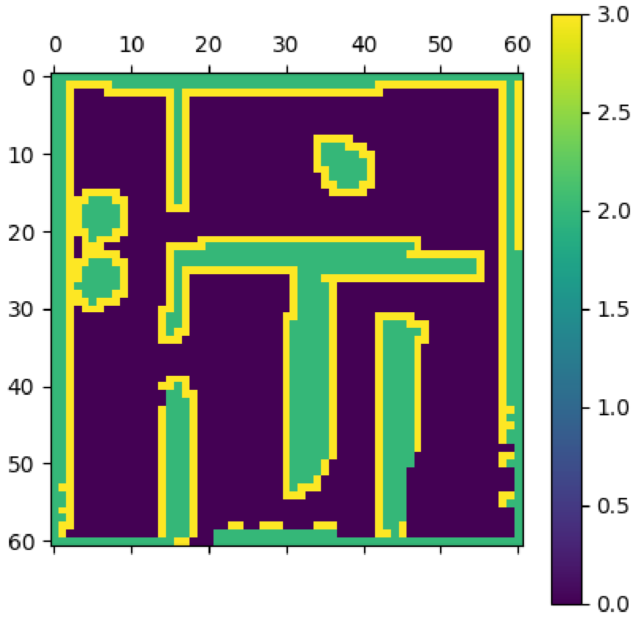

Fig. 4. Final map - particle filter.

IV. Manipulation

For this milestone, the youBot will analyze the states map provided by the navigation part and will identify the center coordinates of the three tables present in it. Thus, the youBot will reach them and take pictures of them to determine which is the target table. Afterwards, the youBot will model each table taking pictures with the depth camera (with two different view angles) and will analyze the points taken through the depth pictures to infer the objects positions and differentiate table 1 and 2. Hence, the youBot will reach table 1, approach it and grab the objects one by one. Each object will be taken to the target table without any falling. The finite state machine controlling the youBot is depicted in Fig. 7, and the logic behind these states is here summarized:

A. *Find target*

1) `searchTables`: the states Map previously saved is converted to a binary file (the obstacles are saved as True) and is analyzed using the package `skimage.measure`. The centroid, their areas and their perimeters are so found and their center is saved as the tables center. A free cell close to the tables is identified and taken as target, the path is so realized by A*.
2) `rotateToCenter`: the camera of the youBot is rotated towards the center of the table.
3) `findTarget`: the camera takes a picture of the tables. The target table is found straightforward, in fact it corresponds to the table where the highest point detected by the sensor is equal to the table height itself. The neighbourhood of the table is defined and the youBot is sent there. The IDs of the tables with objects are assigned and 5 points around the latter are selected. The path is so realized by A*.

B. *Model Table*

1) `modelTable`: the depth camera is used to take picture of the tables in order to identify the position of the objects. Among all the points captured by the camera, only those at a distance between 0.25 m and 1.6 m are kept. This to avoid catching points belonging to the youBot structure and to focus on the table under analysis. A similar methodology is applied to the height of the captured points. Namely, only those samples above 0.187 m are kept.

C. *Image Analysis*

1) `imageAnalysis`: the points collected in the depth pictures are analyzed to infer the position of the objects' center. Specifically, the latter are obtained using the `sklearn.cluster`, that is employed to obtain 5 clusters from the picked points. In Fig. 5 the first table is shown, in Fig. 6 the second one is shown. In order to assign the ID to each table, the distances between the mean objects center points and the objects center points are checked. The table on which the distance is the higher has many space between objects, so it is assigned with the ID 1.

D. *Grasping*

1) `computedestObjects`: the positions where the youBot will be sent to place the objects on the target table are computed. Specifically, the location of the objects is identified dividing 360° for 5 in order to have them evenly apart.
2) `calculateObjectGoal`: the nearest free cell to objects on the table is found. The youBot is then sent there.
3) `preciseForward`: the velocity and the tolerances are set to a low value to have a precise forward command.

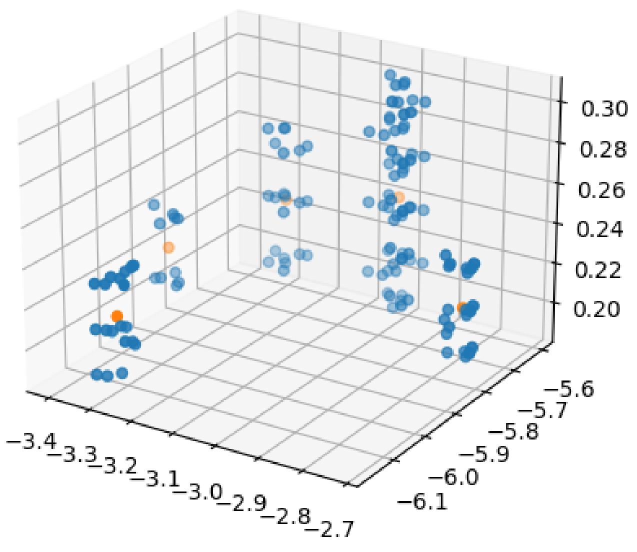

Fig. 5. Centers and objects points on table 1.

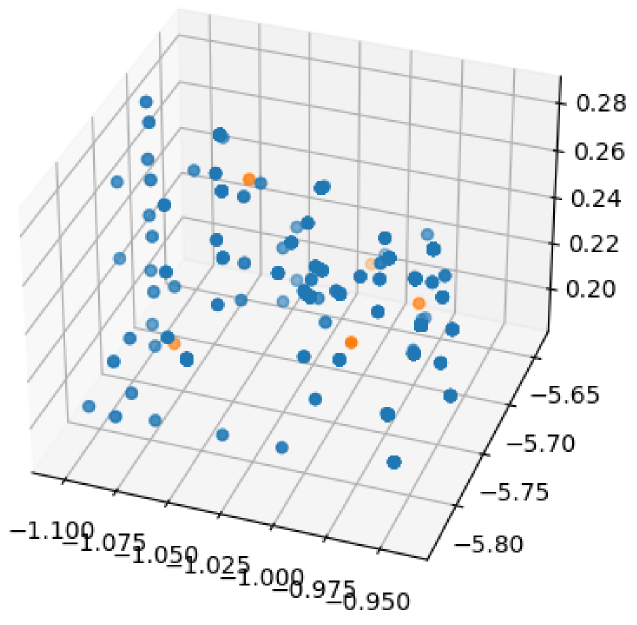

Fig. 6. Centers and objects points on table 2.

4) `rotateAndSlide`: the youBot rotates until it is parallel to the table (having it on the left) and slides closer or further to the table itself. As for `rotate`, the angular velocity is proportional to the angular distance between the youBot angle and the final angle and stops when this difference is smaller than a tolerance.
5) `preciseobjectAnalysis`: the youBot takes a precise picture of the object to grasp to have a quite accurate grasping. The field of view of the camera is set to $\frac{\pi}{12}$. Due to the proximity of youBot to the object, only points at a distance lower than 0.8 m are kept. In addition, the table points are removed discarding all the points below 0.187 m.
6) `armMotion`: the arm takes an object from table 1 or put it on the target table. The target point is computed as the average between the center of the object and the closest point captured by the camera. The turning angles of the arms are obtained from the following system of equations:

$$l_2 + \sin(\phi_2) + l_3 + \sin(\phi_2 + \phi_3) + l_4 = d \quad (3)$$

$$l_2 + \cos(\phi_2) + l_3 + \cos(\phi_2 + \phi_3) = h \quad (4)$$

where l_i represent the lengths of the arm's segments, ϕ_i are the angles, h is the height difference between the object and the gripper and d is the distance between the gripper and the object. Once the arm picks up or releases the object, it goes back to its initial position.

References

[1] M. Petrocchi, "astar_python," Sep. 2018. [Online]. Available: https://github.com/zephirdeadline/astar_python
[2] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 2nd ed. Cham: Springer, 2017.
[3] J. H. Williamson, "pfilter," Dec. 2020. [Online]. Available: https://github.com/johnhw/pfilter

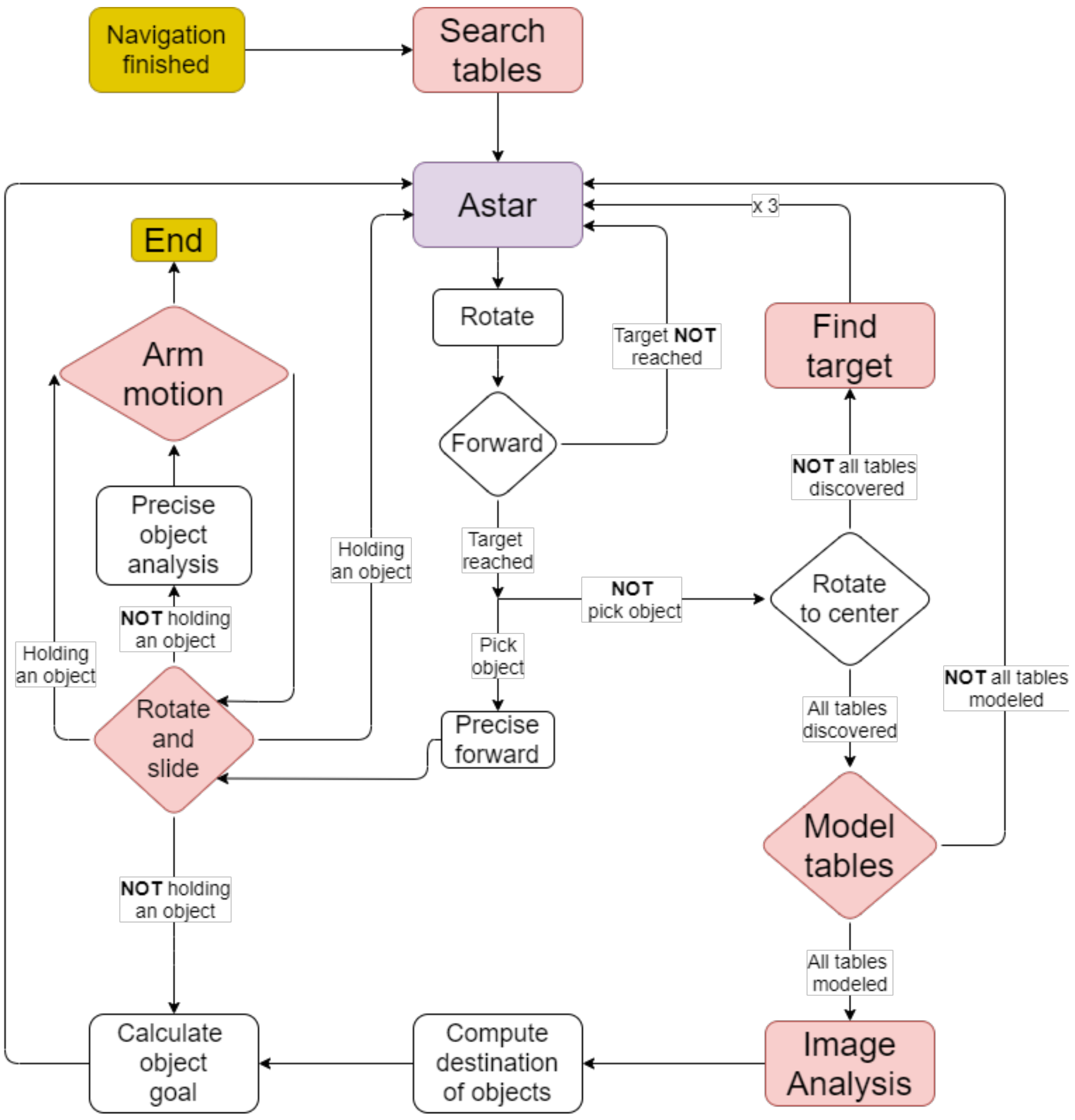

Fig. 7. Final state machine diagram.