INFO0948-1: Final report

Antoine Debor (s173215) Pierre Navez (s154062)

I. Introduction

In this project, a robotic agent called youBot has been programmed. The two mains objectives were to make it able to navigate and build a map with or without accessing its GPS, and the second objective was to program the robot to grab objects from a table and move them on another empty table.

This report presents how these goals have been achieved. In the first part, the navigation problem is discussed. The second one includes the explanations about the manipulation problem. Additionally, the computation times and finite state machines of the respective algorithms are shown.

II. Part 1: Navigation

In this part, the robot starts in an unknown environment that it must fully explore in order to build an appropriate map associated to it. Two implementations are provided. In the first one, youBot has access to its coordinates in the environment whenever needed. In the second case, the robot has only access at its location once at the beginning. It maintains an estimate of its current position thanks to 3 beacons placed at a known location on the environment. A sensor provides the distances between the robot and the 3 beacons and an extended Kalman filter allows to update the estimate of the location when the robot is moving. This filter is presented in II-D. Except this difference, the two versions of the navigation are implemented using the same map representation model, the same path planning algorithm as well as the same controller. The following sections are thus valid for both.

The source code of this part is entitled `myYoubot.py`. Additional modules are defined in other files in the archive folder, they are mentioned in the following subsections. Among them, the Kalman filter can be found in the file `ekf.py`. To run the simulation with the filter, it suffices to add the `-ekf` argument in the command line of a terminal when launching the main source code.

A. 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.

B. 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. 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

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

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.

C. Controller

It has been decided to use a proportional controller to drive the robot. For the forward velocity control, the target 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".

D. Extended Kalman Filter

To computes distances and trajectories, as well as to build the map, the robot must know where it is. In the previous sections, the assumption that youBot could access its GPS was made. In the variant of the problem, the only position known is the starting position. In addition, 3 beacons, which locations are known as well, are placed in the environment. The robot can sense its distance from these beacons through a radio signal sensor.

The tool used to maintain an estimate of its location is the extended Kalman filter. This algorithm is an extension to non-linear models of the Kalman filter presented in 1960 by Rudolf Kalman [3]. Basically, this is an iterative mathematical model which allows to predict the future state of the robot given the current state, some dynamics and the control signal sent to the robot. Moreover, the algorithm maintains a covariance matrix that encodes the uncertainty on the current estimation. Once the predictions are made, the estimate is updated using the information received from the sensors. The algorithm assumes that the motion model of the robot and the information received from the sensors will inevitably contain some noise.

Formally, the state of the robot is defined as a two dimensional vector $\mathbf{x} = (x, y)^T$ that represents its coordinates in the plane of the map. One could have add its head orientation θ but is it assumed that the robot has access to this information at any time. The motion model describes the robot's dynamics, that is, the next state $\mathbf{x}_{k+1}$, given the current state $\mathbf{x}_k$ and an input control signal $\mathbf{u}_k$ at the discrete time k,

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) = \begin{pmatrix} x_k + \delta_k \cos\theta_k \\ y_k + \delta_k \sin\theta_k \end{pmatrix} \tag{1}$$

where $\delta = v \times dt$, v is the linear velocity of the robot and $dt = 0.05$ is the discrete time step. The linear velocity is retrieved from the inverse kinematics of the youBot, some details will be provided later during the explanations.

The Kalman filter hypothesizes that the motion model is always noisy. In fact, equation 1 does not correspond perfectly to the true dynamics of the robot. Inevitably, at each time step, the hypothesis is made that a multivariate random Gaussian noise $\mathbf{w} \sim N(0, \mathbf{W})$ is added to the dynamics. $\mathbf{W} = \begin{pmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{pmatrix}$ is its covariance matrix. After some tests, the standard deviation of the motion noise has been set to 0.05m, which means that the assumption is made that the dynamics model could lead to errors as high as 5cm.

If $\mathbf{P}$ denotes the covariance matrix of the filter estimate, the *prediction step* of the filter goes as follows (the + superscript denotes that the state is predicted):

$$\mathbf{x}^+_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) = \begin{pmatrix} x_k + \delta_k \cos\theta_k \\ y_k + \delta_k \sin\theta_k \end{pmatrix} \tag{2}$$

$$\mathbf{P}^+_{k+1} = \mathbf{F}\mathbf{P}_k\mathbf{F}^T + \mathbf{W} \tag{3}$$

where $\mathbf{F}$ is the Jacobian matrix of the dynamics with respect to $\mathbf{x}$ (in this case, a 2×2 identity matrix).

In order to adjust these previous predictions, the Kalman filter collects some information from the youBot sensors. In this case, these information are some distances between the robot and 3 beacons of known positions. Once again, they are assumed to be noisy. First, a sensor model $\mathbf{h}(\mathbf{x}^+_{k+1}, \mathbf{p}_i)$ is defined, where $\mathbf{p}_i = (x_{bi}, y_{bi})$ is the cartesian coordinates of the beacon i. This model allows to compute what information should receive the robot at the new predicted position. It has the following mathematical expression

$$\mathbf{z}_{k+1} = \mathbf{h}(\mathbf{x}^+_{k+1}, \mathbf{p}_i) = \begin{pmatrix} \sqrt{(y_{b1} - y^+_{k+1})^2 + (x_{b1} - x^+_{k+1})^2} \\ \sqrt{(y_{b2} - y^+_{k+1})^2 + (x_{b2} - x^+_{k+1})^2} \\ \sqrt{(y_{b3} - y^+_{k+1})^2 + (x_{b3} - x^+_{k+1})^2} \end{pmatrix}. \tag{4}$$

Similarly to the motion model, the assumption of a multivariate Gaussian additive noise $\mathbf{r} \sim N(0, \mathbf{R})$, with covariance matrix $\mathbf{R} = \begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$ added to the model is made.

The source code of the youBot indicates that the noise of the sensors is such that $\sigma_{bi} = 0.01$.

The difference between the prediction from the sensor model $\mathbf{z}_{k+1}$ and the observations $\mathbf{z}^{\#}_{k+1}$ from the true sensor is called the innovation ν and will be used to compute the Kalman gain. The following equations are describing the *update step*, which adjust the predictions of $\mathbf{P}^{+}_{k+1}$ and $\mathbf{x}^{+}_{k+1}$:

$$\mathbf{x}_{k+1} = \mathbf{x}^{+}_{k+1} + \mathbf{K}\nu \tag{5}$$
$$\mathbf{P}_{k+1} = \mathbf{P}^{+}_{k+1} - \mathbf{K}\mathbf{H}\mathbf{P}^{+}_{k+1}. \tag{6}$$

In these formulas, $\mathbf{K}$ is the Kalman gain, $\mathbf{H}$ is the Jacobian matrix of the sensor model with respect to the states $\mathbf{x}$ and ν is the innovation. They are defined as follows

$$\nu = \mathbf{z}^{\#}_{k+1} - \mathbf{z}_{k+1}$$
$$\mathbf{K} = \mathbf{P}^{+}_{k+1}\mathbf{H}^{T}(\mathbf{H}\mathbf{P}^{+}_{k+1}\mathbf{H}^{T} + \mathbf{R})^{-1}.$$

These parameters are computed at each discrete time step. After this step, the robot updates its current beliefs and set $\mathbf{x}_k = \mathbf{x}_{k+1}$ and $\mathbf{P}_k = \mathbf{P}_{k+1}$.

Linear velocity computation: In the motion model, the linear velocity of the robot v has to be retrieved from the control signal $\mathbf{u}$. Mathematically, this expression is simply $v = \sqrt{v_x^2 + v_y^2}$. The control signal given as input to the youBot is a tuple of velocities, i.e a forward or back velocity, a side velocity and an angular velocity. However, these are target values. In fact, when the robot receives this control signal, it cannot reach such velocities instantaneously but must accelerate in order to reach them. The raw control signal cannot be fed directly to the motion model of the Kalman filter. Nonetheless, one can compute the kinematics to retrieve v_x and v_y and consequently v. The formulas are derived in the paper [4]. The complete reasoning will not be redone in this report but the idea behind the kinematics is that from a linear combination of the input control signal $\mathbf{u}$, multiplied by the angular velocity of each wheel and a geometric parameter, one can easily retrieve v_x, v_y and ω_z, the latter being the angular velocity but is not necessary in this case.

Overall this filter modelization works well in practice. Indeed, during the map building process, the believed position of the robot on the map is displayed and by comparison to the true map, it seems rather accurate. However, a possible amelioration has been noticed. In fact when the robot reorientate itself or follows a trajectory thanks to its controller, it compares its head orientation with a target angle, the latter being computed as the arctangent of the axis formed by $(y_{target} - y_{robot})$ and $(x_{target} - x_{robot})$ where (x_{robot}, y_{robot}) is the current believed position, returned by the filter. As the filter estimate is noisy and varies a bit at each iteration, even if the robot is not moving, due to the noisy signal received from its sensor, it happens that the head reorientation process lasts longer than it should. A solution has been found in the reorientation state (*cfr.* Figure 1 in the next section), as the robot's forward velocity is nil, the youBot coordinates are locked during all its reorientation, such that the noise in the estimation does not have any effect from loop to loop. This could not be done in the trajectory following state at the target angle is computed even if the robot is moving forward. In the latter case, the robot's coordinates is expected to change from loop to loop.

E. 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.

The same finite state machine applies to both cases, with or without the use of a GPS. In the latter case, when the robot keeps a state estimate, the current belief of the state is computed at each discrete time step. In the two situations, the head orientation of the robot can be accessed without restriction.

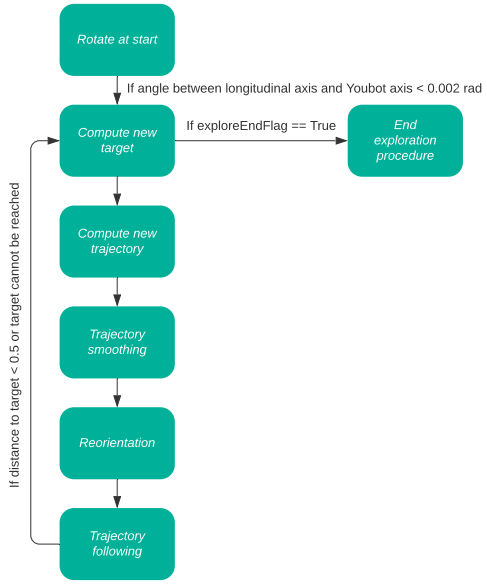

Fig. 1. Finite state machine of the robot's behaviour

F. Computation time

Figure 2 presents the histogram of the computation time of the loops for one instance of the exploration phase in the case where the robot has access to its GPS.

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.

On Figure 3, the same histogram is depicted in the case where the robot maintains an estimate of its current

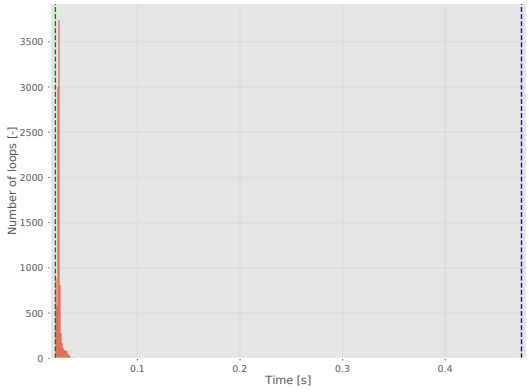

Fig. 2. Histogram of the loops' computing time with access to GPS

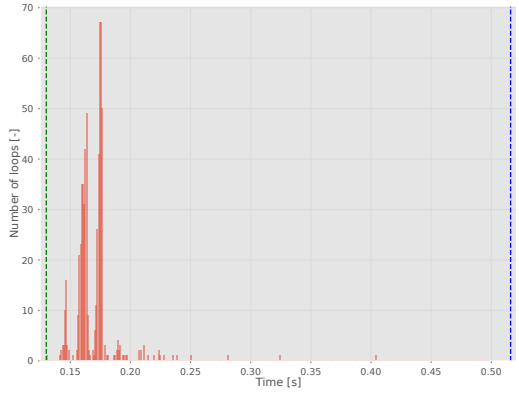

Fig. 3. Histogram of the loops' computing time with Kalman filter

position. This histogram does not correspond to a complete exploration but only to a series of all the state that one can find in the finite state machine Figure 1 that is why the number of loop on the ordinate axis is rather small. One can see on Figure 3 that unfortunately, the loop duration are above the 50 ms threshold needed to match the discrete time step of the simulator. This is due to the fact the at each loop, the Kalman filter has to compute the instantaneous velocities, the equations from the prediction step as well as those of the update step.

III. Part 2: Manipulation

In this part, the robot's goal is to grab some objects lying on a table and displace them on another distant target table. To do so, the robot will make use of its multi-joint arm as well as a series of sensors used for detection and displacement. In the following, every aspect of this task implementation is going to be discussed.

A. Tables localization

Once the exploration phase completed, a map has been generated and can be used to identify the location of the different tables in the room. This approach is based on some image processing algorithm, but another way to proceed would have been to implement a RANSAC algorithm to directly identify the position of the tables during the exploration phase, without need of post-processing.

Fig. 4. Center estimates using `HoughCircles`.

The first approach makes use of *OpenCV* to process the *PNG* image generated from the 2D map obtained at the end of the exploration phase. In particular, the `HoughCircles` function is used to find circular patterns in the map and thus identify table candidates. For instance, using the default map, one can obtain the estimates presented in Figure 4. However, this approach is highly dependent on the fine-tuning of its parameters, and the estimates are not always very accurate. Due to a lack of time, it has thus been decided to hardcode the centers of the tables of interest rather than to improve this technique. We are obviously aware that this is a strong lack of our project, but this has been done in order to have the time to try the grasping part. The notebook used to perform various tests on this technique is provided in the submitted archive as `TableDetectionTests.ipynb`.

B. Object detection

The robot owns various sensors that could be used for object detection. For example, thanks to its RGB camera, the robot could have shot some photographs and use some computer vision algorithm to interpret the images. However, that would be greedy in computational resources and maybe add complexity to the problem. This approach has been followed in the first version of the code, in which one tried to extract the information of interest from a RGB picture using *OpenCV* facilities. In this approach, assuming a picture of the table with an object to grasp in it, the idea was to first erase content below a certain threshold in height and above a certain threshold in depth using a point cloud in addition to the RGB sensor, and then to detect the color of the nearest object in the picture to erase all points in the picture not having this same color. With such a strategy, one was able to extract a picture of the object of interest where each point of the picture not being part of this object appeared black.

From that, it was possible to compute an approximate of the center of gravity of the object. This approach however turned out to be ineffective, and has been abandoned in favour of a point-cloud-only approach, described in the following. Still, the RGB-image processing functions have been kept and can be seen in the file `utils.py` [2].

In the present case, a depth camera is thus used instead. A depth camera allows to take what is called point clouds which are basically a set of tuples $(x, y, z, distance)$ where (x, y, z) denotes the coordinates of a point in the point cloud and $distance$ relates to the distance of the point from the depth camera sensor. Note that z refers in this case to the depth coordinate rather than to the height coordinate. A single point is not a relevant information for the robot, but an ensemble of points can be interpreted geometrically to detect object shapes.

Assuming the youBot is already near a table with objects on it, with the robot's main axis being perpendicular to the youBot-centre of table axis, the protocol for object detection goes as follows. When capturing a point cloud, the depth sensor has an angle of $+\pi/4$ degrees with respect to the robot's reference, hence being able to capture objects which are close the the robot's arm. After shooting a point cloud, the very first step is to post-process the later. Indeed, the height of the tables being known, one can delete points with a height coordinate below this known value, as well as points too far in depth from a given threshold. If this second process is not performed, the robot could identify objects that are actually located at the opposite on the table, hence not being of interest. Once this post-processing completed, one needs to ensure that there are enough points in the cloud. Indeed, the depth camera works to detect obstacles in a range of 5 meters. The room being greater than that, in a specific orientation of the robot, the camera would not be obstructed by objects or not enough for interpretation, such that it would not be possible to deduce relevant information. For instance, if the depth sensor is oriented such that it takes a point cloud of a part of a table and a wall, there will be no remaining points after the previously explained post-processing of the cloud.

Once a cloud filled with enough points is taken, the shape of the objects can be interpreted regarding the interdependence of normal vectors to the points. Indeed, it is supposed that the objects of interest have either parallelepipedic or cylindrical shapes. The strategy is to assume that if the proportion of parallel normal vectors is above 30 percents, the object is a parallelepiped, otherwise it is a cylinder. This strategy has proven to be effective in the considered framework, but could be too naive for more complex-shaped objects.

C. Grasping

The protocol for object detection has been described in the previous subsection. Now, the grasping procedure is going to be explained. It differs a little bit depending on the object shape, the trickiest one being the one for parallelepipedic objects.

Parallelepipedic objects: First of all, even if a parallelepipedic object has been recognized, it could happen that the point cloud does not contain enough faces of the polyhedra, in which case the robot should adjust its position and retake a point cloud shoot. In addition, some of the detected faces could not contain enough points to extract the information needed for the grasping such that one can get rid of them. In both cases, if the effective number of faces is below two, it means that the only detected face is facing the depth sensor, hence being unreachable by the arm, located at the opposite of the robot.

Finally, if the relevant points that last in the cloud are corresponding to two faces, the center of gravity of the set of points corresponding to each face is computed. Considering the two results, the one that is the furthest from the sensor is corresponding to the nearest center of gravity to the robot's arm, and is kept as a target point.

A last thing that the robot verifies by looking at the normal of this target, is that it can move its arm such that it would be aligned in the same axis than this normal vector and so that it would be able to grab the object properly. This is achieved by computing the dot product between the normal vector of the point of interest and another vector linking the base of the arm to the target point. This is not a straightforward operation because the referential of the sensor is not the same as the referential of the robot's arm so at the beginning the coordinates of the normal vector of the barycenter must be moved in the good referential. The two vectors are normalized such that if both axis are parallel, the dot product which corresponds to the product of the norms of the two vectors and the cosine of the angle between the two vectors should be equal to ± 1. As perfect parallelism is not needed, a small error on this value is accepted.

If the considered center of gravity turns out to be a valid one, a small quantity is added to the target point coordinates, which will correspond to the final target point to be grabbed by the robot's arm. Indeed, the robot has to target an approximate of the object's center of gravity, rather than a point on a face, in order to grasp the object properly.

Cylindrical objects: If the object is interpreted as a cylinder, no face handling is necessary. The center of gravity of the point cloud is computed and one adds the radius of the cylinder to its depth value in order to approximate the actual barycenter of the solid, *i.e.* the true target point to be grabbed by the arm.

Object grasping: This process has been divided into several steps such that a proper protocol can be followed whatever the object to grasp. In chronological order, it goes as follows

- First, the arm is moved to an upward position. In this

[2] In the 'IMAGE/CLOUD PROCESSING CUSTOM FUNCTIONS' section.

position it is ensured that no collisions will occur. The base of the arm rotates on its axis until it is aligned towards the target point.
- At this moment, the inverse kinematics mode starts. Before giving the true target point to the arm, a intermediate point located above the actual target is given. In this way, the grasping is not too brutal and there are no risk to knock the object down.
- The true target point is then given to the arm, which places its tip around the object to grasp.
- The signal is then sent to the arm to close its gripper, in order to grasp the object. A counter is used to wait a certain amount of time with the aim of keeping the sequentiality and smoothness of the grasping. If one does not wait for the gripper to close before moving the arm upwards, the object will obviously not be grasped properly.

Without discussing about the robot's controller and its implications on the overall grasping process, the description of the sequence of actions performed by the arm once an object is grasped until it is deposited on the target table is pursued

- The arm is put back in upward position, again to avoid collisions. Then, it rotates with respect to its base's joint until its axis is aligned with the one of the robot.
- Afterwards, the arm is fold back such that the object is not to high from the ground. It has been considered that in real life, this would be smarter to reduce the potential energy of the object which could be fragile and could break easily by falling on the ground. The two upper joints of the arms are moved before the base. [3]

When dropping objects at the deposit location, a similar sequence of actions is followed in the opposite direction. However, the process is much simpler and only consists in moving the arm upwards, rotating it with a fixed angle of $\pi/4$ rad such that it is oriented towards the table, and moving it downwards to reach a height high enough to avoid collision with the table's surface and low enough to cope with the possible fragility of the object again. The object is then dropped by opening the gripper. Finally, the arm is moved to a configuration close to its original one, before the robot goes back to the 'source' table in order to grasp new objects.

D. Controller

The initial controller described in the subsection II-C is kept to handle the path following procedure and control the robot speed. However, another grasping-specific controller has been added in order to deal with the grasping task. In particular, moving from a table to another, turning around a table to shoot point clouds, and place the object at an empty position on the target table are the functionalities added. In this section, the new elements of the controller are described.

As explained in the previous section, the object detection is achieved by taking point cloud shots around the table that contains objects (these shots are taken according to a counter, which is decremented at each iteration, and which is re-initialized when a shot is taken). To do so, the youBot should reach the table in question and place itself in a proper position, such that the camera depth sensor can be used in an optimal way. When the exploration ends, the map contains the tables. Two sets of points forming circles around the tables with objects are maintained (actually, only the easy one is handled). These two sets are basically two concentric circles. The outer one serves as a first target location for the robot, *i.e.* its docking point when approaching the table. When reaching it, the robot is then oriented perpendicularly to the outer normal of the circle it is on, in such a way that, if the depth sensor is considered as the robot's front side, it will turn around the table counterclockwise. Then, its goal is to reach the inner circle designed to be at a reasonable distance from the table to take efficient point cloud and avoiding collision with the table. The inner circle is simply reached by sending a side velocity control signal to the robot, which will stop its slide once it is at a certain distance from the center of the table.

Afterwards, when it is on the inner circle around a table with objects, the robot follows the circle as a trajectory while taking snapshots and processing them as described in the previous subsection. If the point cloud turns out to not be relevant to deduce useful information, the robot does not perform further processing (*e.g.* shape detection) and keep on following the circle and a new snapshot is taken after a little amount of time. This relevance is assessed by checking that, after having deleted too-far and too-low points, the closest point is close enough to the depth sensor.

Assuming the robot has managed to find an object and is grabbing it, its goal is to put it on the target table while paying attention to not make this object fall on the ground. If the grip of the arm is more or less around the barycenter of the object, this is not likely to happen during the displacement of the robot. However, the position where the object is going to be deposited on the target table matters. Naturally, the objects should not be stacked on the target table. A similar strategy which consists in maintaining a set of point forming a circle around the target table is used. A point on this circle will serve as target location for the robot's unloading. Moreover, a counter is maintained to index the different docking points, such that every time a new object is deposited on the target table, the counter is incremented, minimizing the chance to make a heap of object and making them fall. A similar strategy is used for the 'source' table, in order for the robot not to restart at the initial docking point each time it has to grab an object. To achieve that, the set of docking/trajectory points are updated each time an object is grasped, simply by removing the already-visited points from the set.

The approach phase is the same for each table, *i.e.* a lateral sliding towards the table. It is noteworthy that, once the object grasped/dropped, the same move is performed by the robot to go away from the table, and begin its journey to the next table.

[3]Note that, in order to avoid a collision between the object and the depth/RGB sensor, the later is rotated back in its original configuration when carrying an object.

E. Finite state machine

On Figure 5 is depicted a finite state machine of the overall process of grasping. An additional finite state machine describes what is related to the proper grasping procedure that is, how the arm is moving to grasp an object. The latter is depicted on Figure 6.

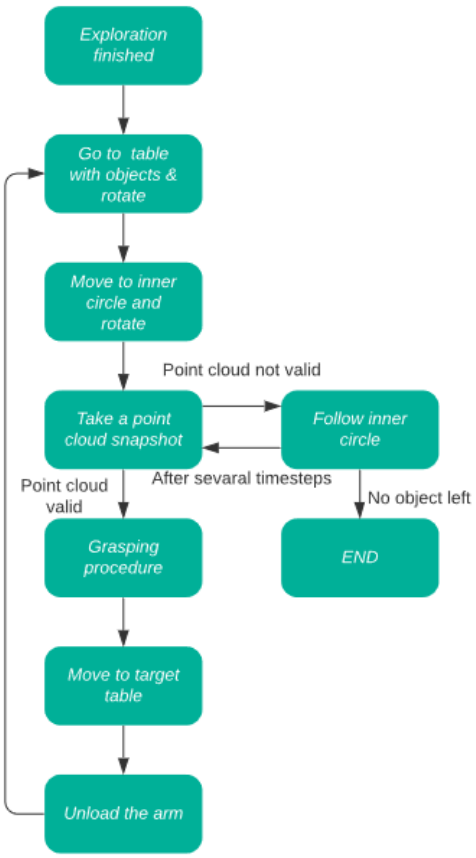

Fig. 5. Finite state machine of the overall process

It is needed to know that none of the finite state machine depicted on the diagrams correspond faithfully to the actual state machine in the code. Indeed, some details have been omitted voluntarily in the diagrams for the sake of clarity. The diagrams only contain the important states of the code. Moreover, the whole program is not really one infinite loop, but rather two ones in series.

F. Computation time

Unfortunately, due to a lack of time, the histogram of the computation time of the loops has not been added.

IV. Video

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

A video of the exploration without GPS with a Kalman filter, commented in the description can be found *here*.

Finally, a video of the grasping phase can be found *here*.

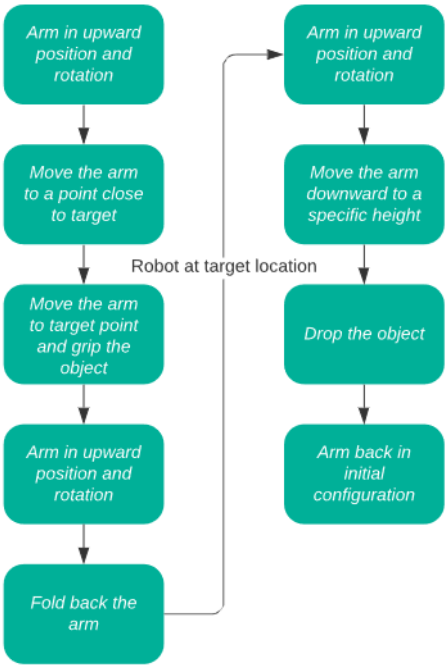

Fig. 6. Finite state machine of arm's behavior during grasping

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.

[3] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[4] H. Taheri, B. Qiao, and N. Ghaeminezhad, "Kinematic model of a four mecanum wheeled mobile robot," *International Journal of Computer Applications*, vol. 113, pp. 6–9, 03 2015.