INFO0948-1: Final report

Bernard Simon (s161519) Klapka Ivan (s165345)

I. Navigation

This section will mainly talk about how we managed to complete the navigation milestone (b).

A. Exploration strategies

For this part of the project, we mainly reused what we did for the milestone (a) where the GPS coordinates could be retrieved at anytime and adapted it to the second milestone where the position estimation was no longer perfect.

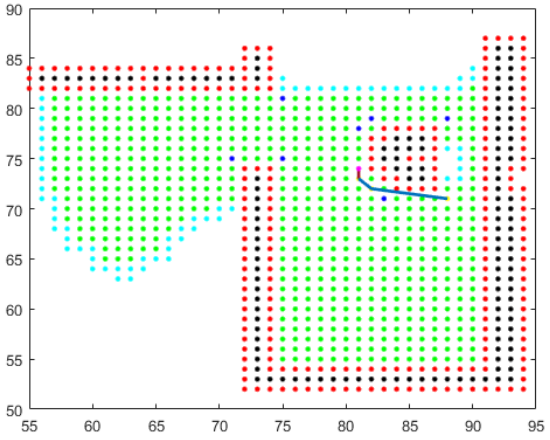

Fig. 1. Map example

The map show above is an example of what the map might look like at the beginning. Each points represents an area of 0.2 by 0.2 meters and each colors has a different meaning :

- Blacks points are the walls
- Green points are the empty explored areas
- Red points are inflated walls, which means area in close proximity to a wall
- Cyan points are the explored points next to unexplored spaces and as such are points that the youbot can go to in order to explore the remaining of the map
- Blue points are the corners of the inflated walls and are used for navigation
- The magenta point is the center of the youbot

In order to update the map, we use the information retrieved by the hokuyo sensor. We first simplify the points returned by the hokuyo too improve computation time, then we make a polygon that represents the cone of vision of the youbot. We update each points inside that polygon as empty explored points and contacts points as walls.

The next step was to find the next location to go to in order to explore the map efficiently. For that we looked at all the cyan points and took the one that was closest to the hokuyo sensor point in front of the robot (the point that is 5m directly in front if there is no obstacle or at the same distance as the wall in front otherwise). This choice might look odd but it increases the chance that the youbot will continue exploring in front of him instead of going back and forth if we had chosen the closest cyan point to the center of the youbot for example.

Once the objective is set, we need a way to avoid the obstacles and for that we used a home made technique that uses the extended corners of the inflated walls. The key principle is that it is safe to assume that these extended corners can serve as waypoints for the youbot to pass through, that they are all interconnected and that all explored points are directly accessible from at least one extended corner. Our algorithm is thus quite simple as it only consists in finding the extended corners and then creating a graph with all of them as well as the objective and the current position. Each point in the graph is connected to an other point if and only if the direct path between them contains no obstacles. We then solve the closest way to the current position to the objective with an Astar algorithm and we thus have a series of nodes (path) to follow in order to reach the objective.

Now that we know from which points to travel to, we need to decided on the velocities. Since we know that we will travel in a straight line, we only need a simple controller that sets the velocities based on the distance and angle. What we did was simply set the forward velocity as the distance to objective times the cosine of the angle between the front vector of the youbot and the the vector to the objective. For the rotational velocity it will simply be set as the angle stated above. We only need to prevent the robot from going backward and now the robot will slow down when approaching its target but also wait for the youbot to turn in the proper direction as well as adjusting itself when it derives from the straight line.

An other solution to this problem, the pure pursuit method, was also considered as it often results in a smoother trajectory. It is not necessarily hard to implement, it consist in setting the objective on the trajectory at a fixed distance in front of the youbot. That way, the robots moves more smoothly as it can adjust early for turns. It may have rendered our exploration a bit faster but we would have had to adjust our extend corners to avoid crashing into walls while doing smoother turns and it would have also made the computation time for the trajectory a bit longer. And since we had already spent way past the 80 hours on this project we did not feel its was necessary and did not justified modifying the way

we handled the exploration.

In order to make the exploration smoother and safer, we implemented a series of states such as a mode for when the youbot is too close to a wall to move away from it. But we also had to add states specifically for the second milestone since now the pose estimation is not perfect anymore. One major problem is that the walls sometimes disappear due to the sensor imprecision and map resolution. To counter this effect in milestone (a) we could just set a wall to be permanent but that solution is no longer possible in milestone (b) due to imprecision and need for correction. So to avoid planning thought walls we added a constraint to recompute the path if a wall suddenly appeared in our way. We also made it so if a path to an explored point cannot be found we set that point to be a wall, since the constraint of permanent wall is gone it does not hurt to wrongly identify a space as a wall.

B. Odometry and pose estimation

In order to obtain information about the position of the youbot we initially thought about using the velocities used in `youbot_drive`. But these velocities turned out to be too imprecise and unreliable to compute a good estimation of the position. So instead we used the rotation angle of the joint of the wheels for our source of odometry. The rotation of the wheels has the advantage of behind precise but has the down side of only giving information about movements that were already done. One must also be careful that measurements of the wheels must be made frequently in order to avoid doing a full rotation between iteration and not being able to see the difference from not moving at all.

The equations used to translate the wheels rotations angles (a_{wl}) into forward (v_f), lateral (v_l) and rotational (v_r) velocities were inspired by equations found inside `youbot_drive` and tuned with empirical parameters :

$$
\begin{aligned}
v_f &= -0.248 * (a_{wl1} + a_{wl2} + a_{wl3} + a_{wl4}) * r \\
v_l &= 0.238 * (-a_{wl1} + a_{wl2} - a_{wl3} + a_{wl4}) * r \\
v_r &= 0.6439 * (a_{wl1} + a_{wl2} - a_{wl3} - a_{wl4}) * r
\end{aligned}
$$

Once we have the velocities we use the velocity model (p.101) from probabilistic robotics [1] to compute the coordinates :

$$
\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v}{\omega}\sin\theta + \frac{v}{\omega}\sin\theta + \omega\Delta t \\ \frac{v}{\omega}\cos\theta - \frac{v}{\omega}\cos\theta + \omega\Delta t \\ \omega\Delta t \end{pmatrix}
$$

Where $v = \sqrt{v_f^2 + v_l^2}$ and $\theta = \arctan(\frac{v_L}{v_f})$, this is done in order to take into account the holonomic motion of the youbot. This model is the best model we managed to build as it achieves an average error of about a millimeter per iteration for the x and y coordinates. One problem of this problem is that it is less precise when it comes to the angle, this is due to the non-linear behavior of the youbot when applying small rotation angles. We tried to find a non-linear approximation of this behavior but to no avail. And since the orientation was available at anytime we use it to compute the angle of the youbot.

Even with the error at each iteration minimized the robot still accumulated an error through the iterations. After a minute of travel the youbot averaged about 10cm of difference with the real position. While it is not necessarily a problem for navigation because the error between iteration is small it can become a problem for building the map. The problem can be easily observed when coming back to a place already visited a few minute before, the robot will not align perfectly with the previous map but will rather make new observations that are shifted by the error. To tackle this problem we synchronize with the GPS every minute in order to guarantee coherence between odometry and real coordinates. In order to synchronize with the GPS we do not simply replace current ones with GPS. What we do is a bit more elaborate, we compute the difference between our last odometry coordinates and the GPS and we use that difference to linearly correct previous observation. This takes a bit of time (about 2-3 seconds) as we need to compute the corrected positions and update our map with all the corrected observations. But this step improves the quality of observations drastically and leads to a significant average error decrease across previous positions.

We initially planned to have a probabilistic approach to coordinates with a Kalman filter and a model for keeping track of the uncertainty of the odometry. This idea became irrelevant since the GPS was the only worthy update to consider and it has no uncertainty. So we never actually needed to have information about the uncertainty of our position since we would never use it.

C. SLAM

The milestone (c) is the milestone we spent the most time on, we did not manage to complete it but we tried several approach and will discuss them here.

The first try we made was to estimate our position only based on the odometry. This gave pretty bad results as straight corridor would be mapped at an angle and as part of the map would begging overlapping pretty quickly. The biggest problem was that the angle needed to be more precise so we tried to use the GPS to correct the orientation and see if it made a difference. The results were greatly improved but the problem of coherence was still pretty huge. Observation of parts of the maps that were explored few minutes ago did not align well due to the accumulation of error in coordinates. The solution for this problem would probably be to implement a loop closure mechanic that would look at previous observation and current one when reaching already visited area and compute the difference in position thanks to scan matching. We did not try to implement it since we had no way to obtain a good estimation of the angle without using the GPS.

The second try was to try to improve the odometry with a scan matching of the current and previous observation. We hoped to improve our angle and overall estimation but the scan matching ended up being less precise than the

odometry alone. So we tried to only take into account the scan matching when it had a high degree of certainty. This did not work very well ether since even with great confidence the scan matching was still less precise that the odometry in most situations.

We then decided to try and use the SLAM built by Matlab [2] as it used scan matching in combination with pose graph estimation. The function worked well for a few iterations but then became unstable and started sending exponentially growing answer without any explanation. We tried to tune the parameters and even gave it an accurate approximation of the position but to no avail. This solution was thus deemed unusable as its behavior was unpredictable.

The next solution is based on the idea that a scan matching of the current and previous observation could still give us useful information especially variance of uncertainty. The odometry variance spreads mostly uniformly in every direction and we hoped that the scan matching could reduce this variance. To picture how it could be useful, imagine that the youbot is traveling in the hallway at constant speed and with what it hopes is a straight line. The odometry will not be sure whether the robots slightly turned left or right but with a scan matching we can figure out the distance between the walls on the side of us. With this information we can make a better approximation of our position, we would have no information of whether we moved forward or not but we could tell that the robot moved slightly to the left or to the right thanks to the difference of distance between the walls to the left and to the right. With this goal in mind, we implemented an extend Kalman filter [1] (p51) for the estimation of the position. The scan matching could output an hessian matrix that once inverted could be seen as a co-variance matrix. We thus went ahead and made a extended Kalman filter that used as an update the results of the scan matching as well as its hessian matrix to have an approximation of the uncertainty. The results were not convincing at all because the scan matching results were still not precise enough and the inverse of the hessian was too random and did not provide a good estimation for the variance of the scan matching. This solution overall would not allow for a good update.

The last attempt at solving the SLAM problem was inspired by this video [3]. We basically tried to replicate what was explain in the video and while doing so we found a matlab implementation of that algorithm [4]. We tried it and looked at the results before trying to implement it for our project. The results in the example seemed really good and would probably be more than enough for our project, the main problem was that the computation for each iteration was about 3 seconds which is way past the 50ms allowed. We knew that even if we managed to adapt it to v-rep we would take way too long to complete the milestones, so we had to gave up on that solution.

While looking in the books from the course for ways to solve this problem we mostly found solutions that rely on landmarks to correct their position [1] [5] . Even in the theoretical course the EKF algorithm presented relies on landmark to work. The problem is that we don't have access to such landmarks in the milestone (c) making those algorithm useless for us. It is still possible to make SLAM of course but it is much more difficult. One option may be to find landmarks yourself but the range would still be limited to 5 meters or we would be required to take and analyze 3d points scan or photos. And since the algorithm from the last attempt was in fact a Phd from a student from the Technical University of Denmark we though that doing something even more difficult might be out of the scope of this project.

The other options we tried to look at were particles filters but the same problems were there. Each particles needed to have its own map which would take a lot of time to compute and for good results we needed a large number of particles making it almost impossible to run it below 50 ms. The problems were also that we needed a way to give a score to each particles in order to re-sample them and we could not find a way to score them effectively.

Overall we spend over 50h just to look at different possibilities and ideas for solving the SLAM problem without even taking coding into account. We learned a lot from this research but never managed to find a way to complete the milestone (c) so we are really curious on how it can be solved and hope that you will be able to show us how it is done.

II. Manipulation

A. Table identification

The first step of the table identification is to find their approximate position on the map. To do so, we use our occupancy map where we increase the resolution, then we apply the two stage circular Hough transform (thanks to `imfindcircles`). After some image filtering operation (such as Canny's edege detector), the goal is to find, at fixed radius, the local maxima in the Hough parameter space for the center position. Then we look for the best radius for those maxima. The algorithm also include a `sensitivity` parameter ($\in [0.0, 1.0]$) which determine the threshold at which a maxima is considered as valid. As we want to ensure to have 3 tables found. We may need to play with this parameter. We have determined a value that often start well to begin with (0.88). If this value give too few table, we increase by a constant increment the sensitivity, and our research space becomes $]0.88, 1.0]$ (if we have too many, we decrease the sensitivity and our space becomes $[0.0, 0.88[$). While we do not have 3 tables found exactly, we continue our procedure with the update of the upper/lower bound of the research space. If our increment bring us out of the research space, we divide the increment by 2 to refine the search. If our increment becomes too small, we give up the search.

Now that we have the approximate center of the table, we have to determine an objective close to the table. To do so, we build the vector from the table to youbot and determine the point that is 60cm away from the side of the table on that vector. If this point fall in obstacle or in inflation of obstacle, we apply a small rotation to the vector to find another close point (this procedure can be repeated several times if needed). If the objective fall in obstacle/inflation

during the travel, because of an hokuyo update or a GPS resize, a new objective to approach the table is determined the same way.

Once we are close to the estimate center of the table (and face to it), we do what we call a "triple scan". It consists in three contiguous scan taken from the same position. It allows to cover $3\pi/8$ of opening angle with 3 times as much ray as a $\pi/8$ opening angle. The 3 scan are linked by using a rotation matrix on the left and the right scan. We also add a `remove_wall` feature that detect the points above a threshold and erase all points below those point in the 3d point cloud. From our 3d triple scan, we isolate the table layer and project it by removing the height. We then sort them in azimuthal direction and take the leftmost, righmost and middle point. With these 3 points we can easily compute the center of the table by building 2 chords and finding the intersection of their bisection (the radius computation is straightforward knowing the center and any point of the circle)[1]. With this new center position we can approach and face more precisely the table.

When we face the table we make another triple scan, but this time we isolate the object layer (still removing height dimension). We know need to distinguish the tables. The empty table is simple to identify as there is no object in the projection. Then for the easy and hard table, we know that objects on the hard table are packed while the ones on the easy table are spread. So we compute the variance in both direction and classify table above a threshold as easy (objects spread have a high variance in position), while table below the threshold is identified as the hard one.

B. Grasping strategies

After the table identification, the second task is to carry the objects from the easy table to the empty table.

The first part of this approach, going back to the easy table, is similar to the table identification : face the center estimated on the map, estimate a better center with a triple 3d point cloud and move youbot to a given distance to the table.

Once the initial approach is done, youbot rotate by 90° to the right and slide closer to the table. Youbot then corrects its distance to the table and its tilt to the tangent thanks to several 3d points cloud. The goal is to "anchor" youbot to the table, the next motions will be circular motions around the table (while keeping the center of youbot at a given distance and the main axis of youbot always aligned with the tangent).

The next step is to find an object to grab, to make sure we can see all the table we make a triple scan pointing to the center of the table. Using trigonometry in right-angled triangle, we can compute the angle needed for the sensor to do so thanks to :

- the distance from the center of youbot to the center of the table

[1]For this purpose we use a function from the file exchange platform of matlab which provide a robust computer implementation of the algorithm `https://ch.mathworks.com/matlabcentral/fileexchange/57668-fit-circle-through-3-points`

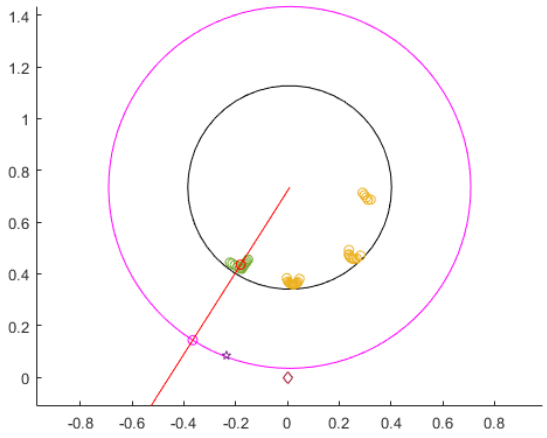

Fig. 2. First scan of the table : the ⋆ is the youbot's center and the ◊ is its sensor.

- the distance from the center of youbot and its sensor
- the angle between both previous vector (right angle as youbot is aligned with the tangent)

With our 3d scan, we first determine the relative position of the table, then we make a projection (removing height dimension) of the points that are higher than the table. Once the objects' points are isolated, we first search for the closest to youbot's center, then look for the other points close enough to this closest point to be in the same object[2]. With the points of a single object, suppose to be the closest one, we compute the approximate center of mass with those points then determine the rotation needed for youbot to align with its side with the object. Note that we also compute an estimation of the object to the objective which will be useful later. The figure 2 allow to visualize this step.

Thanks to the equation of the circular motion[3] :

$$v = r \cdot \omega$$

We are able to drive rather precisely around the table. As we drive in a circular motion, rotating by a given angle around the table can be controlled thanks to the absolute orientation of youbot (by rotating by 10° in the circle, the absolute angle of youbot also rotate by 10°). Once the rotation is completed, we make another correction as done above after sliding close to the table, this should ensure the rightness of the distance to the object estimation computed before, which will be used for the next scan.

Now that we are next to our object, we make a better scan of the object. The angle of the camera is determined the same way than for the center of the table, except that here we use our estimate of the distance between actual center position (what was the objective) and the object rather than

[2]As we know that the shape are about 5cm width and not too close to each other this work well, this would however not be enough precise for the hard table

[3]In our case, using youbot input velocity, this equation was true with a multiplier factor determined empirically

the distance to the table's center. We also reduce the opening angle of the camera to $\pi/12$ to enhance the information we have on the object. Note that the triple scans have a slight numerical error at the junction between the scan which would be annoying for the object identification.

On this new scan, we first isolate the table layer and object layer and restricting the object layer to the closest object the same way as before the circular motion. We know need to identify the shape of the object thanks to those points. To do so we consider 3 different cases (2 for the square and 1 for the circle). Note that for the square we assume that the width is 5cm (as the grip could not however grab a wider object)

Case 1 : single line. The first step is to check whether or not we only got a single line from the scan, which would correspond to the side of a square. The check is done by computing the correlation coefficient of the point, if they are above a enough restrictive threshold, we have a line. In this case, we extrapolate the center 2.5cm beside the center of this line. We can then build the two perpendiculars to the sides of the square than cross the center. With those lines we can find the point, on the circular trajectory of youbot, that is the closest (to the object) and aligned with a side of the square-base. We also have enough information to pre-compute what would be the angle and the distance from this new objective to the object.

Case 2 : two lines. If we do not fit the first case, we try to find 2 lines. Note that from the sensor point of view, once the points are sorted in the azimuthal direction, all the points from each lines are grouped on the same side. We thus try to find at which point we have all the left point point belonging to a line and all the right one belonging to another. To do so we compute the correlation coefficients of the points for each side and accept if both lines are above our threshold (note that we need at least 2 points for each line). If a match is found, we get the intersection of those lines to get the corner, then apply the vector of those lines (with a length of 2.5cm) to get the center. The next steps are identical to case 1. The figure 3 allow to visualize this step.

Case 3 : circle. We assume that if none of the previous case fits, we are in the circle one (our actual youbot is not made to identify that a shape is unknown). In this case we find the center and the radius with the same function as for the table. Then we distinct 2 sub case for the youbot following behavior. If the object is already close enough to the arm, we compute the direct angle and distance to grab it. If it is too far, we define as objective for the center the position on the trajectory that is crossed by the radius of the table that go trough the center of the circle, compute the distance and angle for the arm based on this position, and then follow the same step as the square-based objects. The figure 4 allow to visualize the direct grab step.

The three next steps are not performed for the direct grab of the circle-based objects.

The next step is first to reach the new objective with a circular motion. Once the center is well placed, we move forward by 0.162m which is the distance between the center

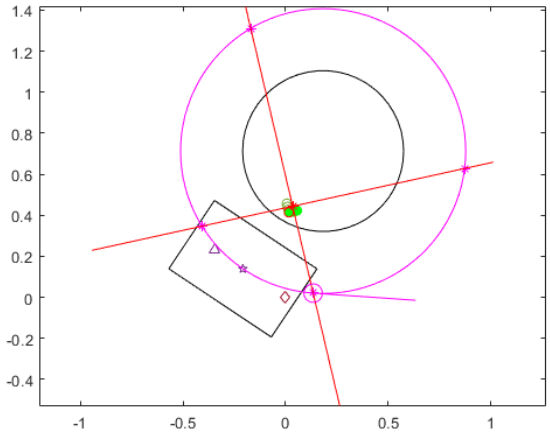

Fig. 3. Second scan of the table : the $\star$ is the youbot's center, the $\Diamond$ is its sensor and the $\triangle$ is its arm.

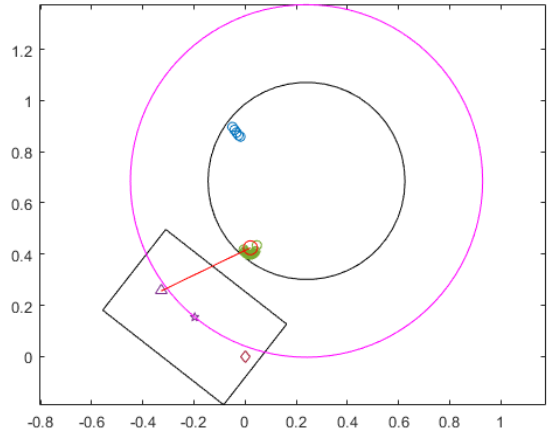

Fig. 4. First scan of the table : the $\star$ is the youbot's center, the $\Diamond$ is its sensor and the $\triangle$ is its arm.

and the arm in youbot. By doing so we align our arm with the object.

At this moment, we could use the distance and angle computed before to grab the object. However, it appeared that the small accumulation of error (estimation of the center of the table, circular and forward motion, ... is a bit too much to grab the objects reliably. We thus need a final scan to make final adjustment (also at $\pi/12$). Here again we have to compute the angle for the sensor. However this time we do not have a right-angle anymore so we need to use the law of cosines (using the grab distance, grab angle and distance between sensor and arm) to get that value. This also allow us to have an estimate of the distance between the object and the sensor that will be useful afterward.

For the final adjustment we will only use forward and backward motion, this avoid to add the error on the estimation of the center of the table to our problem, plus the circular motion is a bit more complex than the forward/backward,

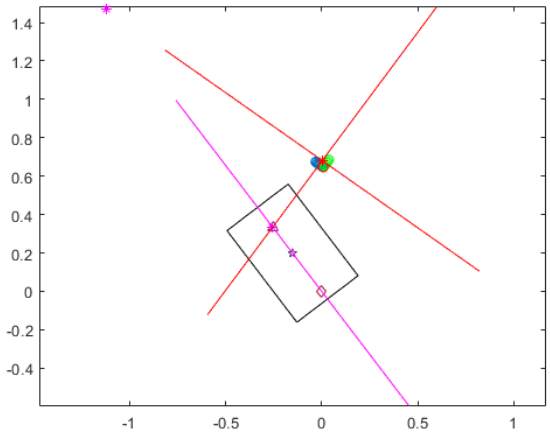

Fig. 5. First scan of the table : the ⋆ is the youbot's center, the ♢ is its sensor and the △ is its arm.

thus probably also adding a bit of error. The new considered trajectory of youbot is now a straight line, and we compute the intersection of this trajectory with the perpendiculars in the case of the square to compute the forward/backward correction. In the case of far circle-based object, we keep our actual position. From this position (corrected for square and actual for circle), we compute the final distance and angle to grab the object. After that step (or after the last forward/backward correction for the square-based) we can start the grasping. The figure 5 allow to visualize the direct grab step.

The direct grab for the close circle-based object resumes here.

Now that we are well placed and that we know the grab angle and the grab distance we can perform the grasping. The first step of the grasping consist in placing the arm at the level of the objects and directed to the object (i.e. the grab angle). Note that the arm is more limited (in the length it can achieve) in is its front direction than by using it backward. Thus, if the grab distance is above a threshold, we use the arm in the reverse direction. We do not use it in reverse every time because it also have a higher minimal distance, thus it can hit objects before grasping them if they are too close. Once the arm is aligned, we use the inverse kinematic mode to extend the arm up to the grab distance. When the object is grasped, we block it against the tray of youbot to ensure its stability during the travel.

The next goal is to go back to the empty table. The procedure is the same as for the easy table : approach the table, find a more precise center, align, and finally "anchor" to it (rotate, slide and correct).

We now need to find where we can place the object. The first object is placed where youbot anchor the table. When we must place the second object, we remember the orientation of youbot when the first object has been placed and go to the location around the table where the orientation is equal to the first one plus a constant number of degree. For the following ones, we just add as many times this constant to the initial placement as there are objects already placed.

Once youbot is well placed around the empty table, youbot go forward by 0.162m (distance between center and arm) and place at 90° the object with a placement script made by experiment.

The youbot can finally go back to the easy table to grasp another object, the loop end when the first global scan of the table does not give any object.

Link for the video of the project [6].

REFERENCES

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. Cambridge, Mass.: MIT Press, 2005. [Online]. Available: http://www.amazon.de/gp/product/0262201623/102-8479661-9831324?v=glance&n=283155&n=507846&s=books&v=glance

[2] Mathworks, "Slam," Jan. 2020. [Online]. Available: https://www.mathworks.com/help/nav/slam.html

[3] Cyrill Stachniss, "Grid-based slam with rao-blackwellized pfs," Dec. 2013. [Online]. Available: https://www.youtube.com/watch?v=U6vr3iNrwRA&list=PLgnQpQtFTOGQrZ4O5QzbIHgl3b1JHimN_

[4] Adrian Llopart Maurin, "rbpf-gmapping," May 2016. [Online]. Available: https://github.com/Allopart/rbpf-gmapping

[5] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 1st ed. Springer Publishing Company, Incorporated, 2013.

[6] Ivan Klapka, "Projet robotics 2020," Aug. 2020. [Online]. Available: https://youtu.be/BDDgoceRufo

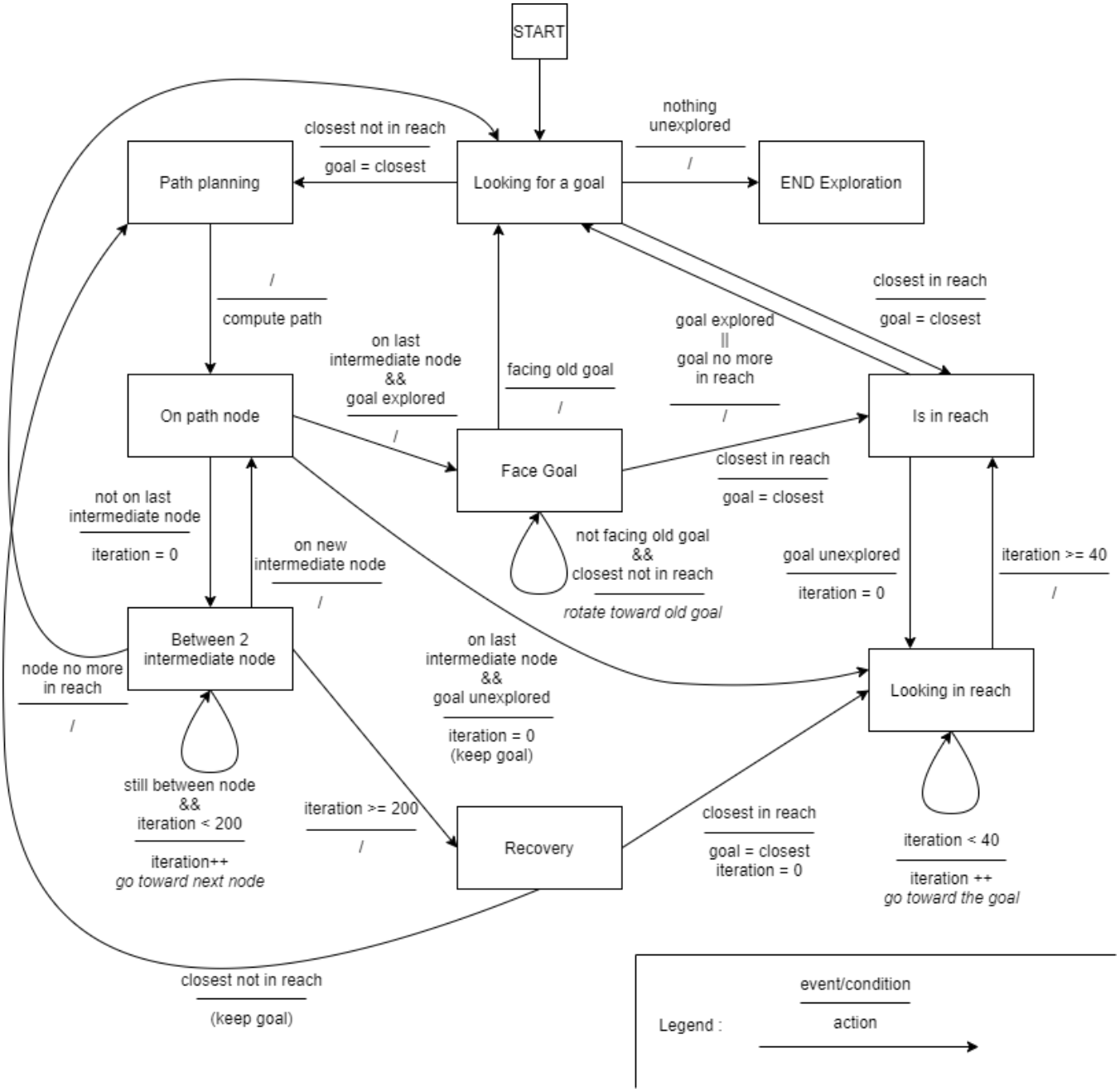

Fig. 6. FSM of the exploration, GPS update is done every minute in addition

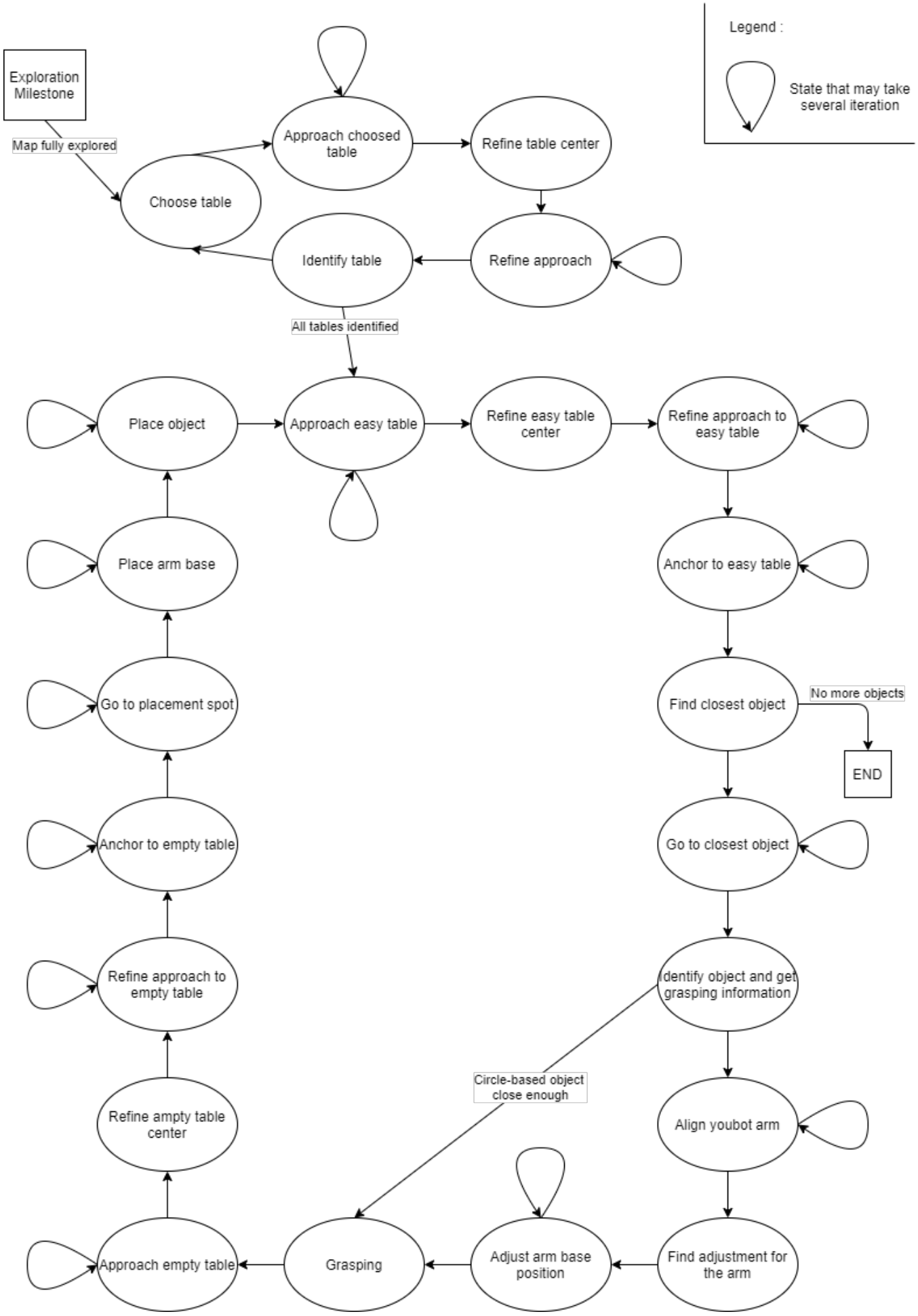

Fig. 7. FSM of the manipulation, traveling detail follow the same principle as the exploration (without goal redefinition adn inReach idle looking)