INFO0948-1: Final report

Maxime MEURISSE (s161278)

Valentin VERMEYLEN (s162864)

I. INTRODUCTION

The goal of this project was to learn how to manipulate a robot (a *youBot* from KUKA) equipped with some sensors (Hokuyo and RGBD), represented in Fig. 1, in order to automate simple tasks.

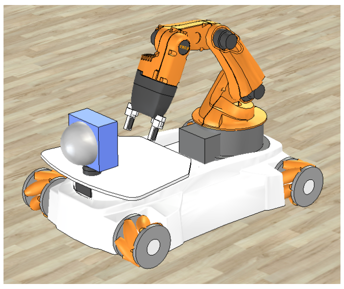


Fig. 1: The youBot robot used for the project.

The first step was to navigate in an unknown environment (but of known dimensions) to produce a precise representation that could be used later (typically a two-dimensional map). This task, which we will call "navigation" later on, is fully explained in section IV.

An example environment for the robot is shown in Fig. 2.

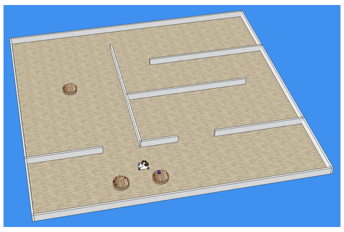


Fig. 2: An example of environment for the robot.

The second step consisted of transporting, using the robot arm, objects from one table to another. This task, which we will call "manipulation" later, is fully explained in section V.

Both tasks must be performed completely autonomously by the robot, regardless of the environment in which it is located. No human intervention shall be necessary.

To carry out this project, we used the V-REP simulator (from Coppelia Robotics) to simulate the youBot and its environment. We used the MATLAB language and several of its toolboxes to program and plan the robot's actions.

Our implementation allowed the youBot to successfully perform navigation and manipulation. A video showing the robot in action is available via the following link:

`https://youtu.be/DcAZKloe3Qk`

This document will begin by briefly explaining the organization of the code. Then, it will explain in a general way, via a diagram representing the finite state machine, how the robot plans its actions. Each milestone (navigation and manipulation) will then be detailed: strategy used, problems encountered, etc. Finally, this document will end with possible improvements to our implementation and a conclusion.

II. CODE ORGANIZATION

We have chosen to work mainly with the *object-oriented* paradigm. We have therefore identified the main elements and implemented them through objects:

- an object `MapManager` to handle the representation of the environment;
- an object `RobotController` to handle the robot.

Our code is organized like this:

- a folder `classes/` that contains files `MapManager.m` and `RobotController.m` mentioned above;
- a folder `imported/` that contains all the functions we found on the Internet. The source of each function is mentioned in the comments of the file;
- a folder `mat/` that contains representations of the environments the robot has explored (in the form of exported MATLAB variables so that they can be saved and reloaded later) and other resources;
- a file `main.m` which is the main file of our implementation;
- a file `manipulation.m` that contains the implementation of the "manipulation" task;
- a file `navigation.m` that contains the implementation of the "navigation" task;
- a file `startup.m` to prepare MATLAB for simulation.

To run the code, first configure the `trs/` folder (as explained on the official project website) and attach it to the code. Then, run the `startup.m` file (by default, MATLAB should run it automatically at startup). Finally, just run the `main.m` file. The latter can be edited to change the dimensions of the map, the time in seconds between two scans correction, and other parameters.

We have tried to make our code as simple, readable, and understandable as possible. We have exploited the object-oriented paradigm to avoid code repetition and to encapsulate the actions of each element neatly. So we have a very simplified interface for the robot and the map (e.g. `robot.move(objective)` to move the robot to the objective `objective`, `map.show()` to display the map, etc), which greatly simplifies our life afterwards for the use and maintainability of the code.

III. Finite state machine

In order to accomplish both tasks, the robot performs a series of actions in a specific order. All these actions are summarized in the finite state machine shown in Fig. 11[1], in the appendix.

The robot states for navigation are implemented in the `navigation.m` file. The robot states for manipulation are implemented in the `manipulation.m` file.

Since the simulator works with a 50 ms time step, the main difficulty is to make each action of the finite state machine execute within 50 ms. We have optimized our coding as much as possible and, in general, it runs well under this constraint. Only the actions like the movements of the robot arm take more time. For the latter, we stop the robot to avoid inconsistencies in its movements due to the time lag with the simulator.

We noticed that the rendering of the map took a lot of time (about 20 ms for navigation and about 40 ms for manipulation). So we decided to update the map only every 50 iterations for navigation. Concerning the manipulation, we believe that the map, not evolving over time, is no longer necessary. We are therefore no longer refreshing it (except for few rare exceptions).

IV. Navigation

As shown in the finite state machine (Fig. 11), the robot performs a series of actions to successfully navigate. These series of actions are very similar for each difficulty (milestones 1a, 1b and 1c). Only the action "update position and orientation" varies between the different difficulties.

Each of the actions will be detailed to explain how it works and to justify our choices of implementation.

A. Initialization

The initialization phase consists of determining the initial position and orientation of the robot. When we have access to the different sensors, we simply use them. When we don't have access to the sensors, we set ourselves an initial position and an initial orientation for the robot (in our case, we decided that the robot starts in the center of the map). Even if this position does not correspond to the real position of the robot (the one returned by its sensors), it does not matter: we build the representation of the map only based on the movements made by the robot; it will be built entirely with respect to the initial position set.

As we do not know where the robot starts, but we know the dimensions of the environment (15 meters by 15 meters), we have created a representation that is twice as long and wide as the size of the environment. This avoids getting negative positions, which is annoying when working with matrices. When we don't have access to the sensors, we therefore set the robot to start at the center of the environment representation, i.e. at point `[15, 15]`.

When the initial position and orientation are set, we mark the robot's neighborhood as free positions. Indeed, the position of the robot represents a single point on the map, but the robot itself counts for several points since it has a width and a length. In order to take into account all the points that the robot represents, we take into account its neighborhood.

[1]The figure is in vectorial format, do not hesitate to zoom in on it for better visibility.

B. Update position and orientation

This step, and all subsequent steps, are repeated at each iteration. Updating the position and orientation of the robot is done differently depending on the difficulty.

1) Permanent access to sensors: When we have constant access to sensors, we simply use the robot's GPS to update its position and orientation. This operation is very simple and instantaneous.

2) Odometry and GPS correction: For the second navigation milestone, we only have access to the GPS data every minute. The approach we have thus taken is to estimate the position and orientation of the robot at each time step via the odometry.

When looking at the function `youbot_drive`, we have seen that the angular displacements and rotation between two-time steps can easily be retrieved via a linear combination of the movement of each wheel. Multiplying these values by the radius of the wheels of the robot (5 cm for the displacements and an experimental value of about 12.75 cm for the rotation), we can get the linear displacements and the rotation in radian. However, the rotation has finally been estimated thanks to the orientation sensors, since it is available at any time. The value we obtain with the linear combination is discussed in our SLAM attempts section.

Plugging these values in the following equations, taken from [1] (v is the forward velocity during the time step and w the rotational one),

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

we retrieve correct approximations of the displacement of the youBot along both axes. However, we have a slight deviation between the estimated position and the real one. In order to correct that, we could have gone for a particle filter, however we do not believe it would have lead to a better solution that the one we have chosen.

We have realized that the deviation from the estimated position to the real one evolves in a quasi-linear fashion in our 60 s runs between two corrections. In order to correct our belief about the map, we have thus decided to simply compute the deviation between the estimated position and the real one at the correction phase, to divide that number by the amount of scans done in that time in order to get the increment of position to correct between two scans, and to correct these scans by modifying the position of the youBot accordingly. The result of these scans are then put in a map that holds only corrected values, and from which the running map is copied after every corrections.

This does not yield a perfect result every time, but we obtain a map that is usable without problems in the manipulation step.

3) SLAM: In order to implement the SLAM, we have tried numerous things, but none has provided reasonable results.

We have based ourselves on the odometry for the estimation of the youbot state, but we found no ways to reliably link the odometry to the rotation, whereas we had almost no problem doing so for the forward movement. This certainly comes from the physics constraints of the simulator and the fact that there is a lot of friction and slipping for low angles of rotation.

Thus, with that highly imperfect prediction of the rotation, we decided to try correcting every few steps with some scan matching, as the SLAM algorithm of MATLAB is supposed to do, but we reached no success and had a very bad map from the first seconds. We also tried using the MATLAB SLAM facility but it also lead to very bad results. We also tried implementing our own particle filter, using the odometry equations as state transition function and adding Gaussian incertitude to the values of v and w, but it lead nowhere. Since we had no way of having a real measurement to correct ourselves, it is not very surprising. The estimation given by the wheels was more precise than any measurement for the displacements, but nothing was precise enough for the rotation. Our only other mean of getting a measurement was through scan matching, and it was not precise enough. We did not try to implement a Kalman Filter since we thought it would not provide better results, and our time was running short.

Talk about relaxations, and the fact that we don't have beacons nor their positions nor their numbers, and even so we would deviate in the snake part since there is no beacon there. -> SLAM is designed to fuck us.

The estimation of the w based on the odometry, however, is far from perfect. Indeed, the relationship that exists between the linear combination of the angular movement of the wheels and the real rotation is semi-linear, but with high variance for small angles, as can be seen in Fig. 3.

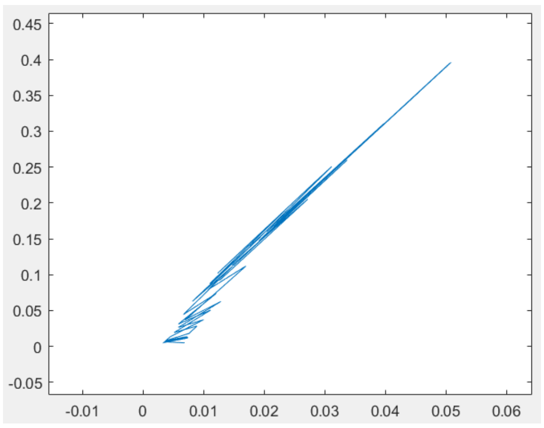


Fig. 3: Relationship between the linear combination of the angular movement of the wheels and the real rotation.

Since we have access to the real orientation at all times, it is actually not necessary to estimate the orientation based on the odometry, but it was done in the hope we would get a working SLAM.

C. Update data from Hokuyo

This stage consists in recovering the various points captured by the Hokuyo. We differentiate between free points and contact points and then transform them into absolute coordinates to insert them in our representation of the environment (by default, the points are in coordinates relative to the robot).

The transformation of all the points of the Hokuyo is quite expensive. We used a function, found on the Internet, which simplifies the polygon formed by these points in order to accelerate the process considerably while preserving the appearance of the polygon.

The free points are inserted as such in our map, and the contact points are inserted as walls in our map. It is therefore this step that updates our representation of the environment.

Our implementation, and especially the coordinate transformation, is strongly based on the one given as an example on the GitHub of the course.

When we don't have access to GPS constantly, we save the different data from the Hokuyo at each iteration in order to correct them later.

D. Obstacle nearby

When the robot is moving, it can sometimes move towards an obstacle without knowing it (because initially, this obstacle was not yet discovered on the map). To prevent the robot from rushing into it, we have implemented a mechanism that allows it to stop and start again.

To know if an obstacle is near (and in front of) the robot, we select several frontal rays of the Hokuyo and calculate the distances between the robot and the contact points of these rays. If only one of these distances is less than a threshold (which we have set at $50\,\mathrm{cm}$ from the center of the robot), we immediately stop the robot. We reset his objective so that he determines a new one and starts again correctly.

When an obstacle has been detected, we no longer perform the check for 50 iterations to allow time for the robot to move. Indeed, if the robot is close to an obstacle, it would constantly be considered as such if we check at each iteration and would therefore remain blocked for an infinite time (since it would stop at each iteration and would not have the time to move).

E. Objectives and path finding

To move the robot, we use a goal and path system. We can distinguish two types of objectives: the current objective and the final objective. The current objective is the next point to which the robot must go. The final objective is the endpoint to which the robot must go.

So the first time we calculate a final objective for the robot. The latter is simply the closest unexplored point of the robot on the map. In order to avoid choosing unreachable points, we check that the found point is surrounded by at least one free (thus reachable) point. When this point is determined, we calculate a path to it using the A* algorithm. Once the path is obtained, we determine the robot's next point, its current objective, by simply taking the first point of the path (and removing it from it).

In the next iterations, we check on the robot has a current target. If it does, then we don't do anything more, we let the robot go there. If the robot no longer has a current objective, we choose a new one by taking the first point of the path. And if the path is empty, i.e. the robot has reached its final goal, we recalculate a new next point to explore, and the cycle starts again.

We consider the navigation is finished when no new point to explore can be determined. Indeed, in this case, it means that all points have been explored. We had the initial idea to use, in addition to this technique, a percentage of exploration: we would stop the navigation when the robot had explored more than 99.5% of the environment. However, due to numerical errors, especially when we don't have constant access to the sensors, this percentage was distorted and exceeded 100%. So we dropped the idea.

To plan a path, we used the A* algorithm. We found a MATLAB implementation on the Internet and used it. The choice of this algorithm came naturally to us because we had studied it before and its performance is very good. Indeed, it operates under $50\,\mathrm{ms}$, which respects the constraint of this project.

However, we have that its execution time is proportional to the size of the environment. To overcome this problem, we thought of using a solution where we define key points on the map, and that we plan a path while still passing these key points. This would drastically reduce the execution time since the algorithm would only act on a few points instead of the whole map. However, with such a solution, the definition of the key points is crucial: if the choice of key points is wrong, the robot may not find a path. So we finally dropped this method to keep the traditional A* algorithm, but we are well aware that on a much larger map, the $50\,\mathrm{ms}$ threshold may be crossed. For safety reasons due to this threshold, we stop the robot when calling the algorithm.

We also tested the D* algorithm provided with P. Corke's Robotics toolbox, but it was too slow so we dropped it.

F. Move the robot

To make the robot move, we use its forward-backward and rotational velocities. In order to make the movement smooth, we make the robot move forward and turn at the same time. The challenge is therefore to choose good values for these velocities.

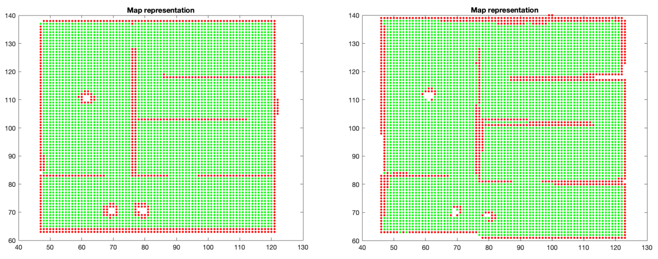
The first step we do is calculate the angle (in an absolute frame of reference) that the robot must have to be aligned with its objective. We then calculate the distance between this angle and the current angle of the robot (the third Euler angle of the robot). We also calculate the distance (L1 norm) between the robot's position and its objective.

The second step is to assign values to the velocities. We distinguish several cases depending on the distance between the angles. If the latter is very high, the robot rotates and does not move forward. If the latter is moderately high, the robot turns and begins to move slowly. If, finally, the distance between the angles is very small, the robot moves forward and does not turn. This fairly simple system makes movement fluid and ensures that the robot does not deviate from its path.

G. Navigation finished

When the navigation is finished, we obtain a representation of the environment where the free (accessible) points are represented in green and the inaccessible points (walls, tables, etc.) are represented in red.

When GPS is used at each iteration, the representation of the environment is very good and very precise (Fig. 4a). When the GPS is not used (or only a few times), the representation is inevitably less precise, but quite usable for the future (Fig. 4b).



(a) With GPS. (b) Without (or few) GPS.

Fig. 4: Representations of the environment at the end of the navigation.

When we have access to the GPS once every minute, we found that waiting for the last update at the end of the navigation allowed us to get a better map for manipulation. So we implemented a mechanism that, when navigation is finished, waits for the last correction with GPS before continuing (so, the robot doesn't move and wait, for a maximum of 1 minute).

V. Manipulation

As illustrated in the finite state machine (Fig. 11), the robot goes through a series of states to perform object manipulation. These states are globally identical for all difficulties. Each state is detailed in the rest of this document.

A. Table detection

The first step of the manipulation consists in detecting the position of the tables on the map representation obtained with the manipulation.

To do this, we use MATLAB's `imfindcircles` function. This uses a *circle Hough transform* to detect all the circles on an image. The image considered here is the occupancy matrix corresponding to the obtained occupancy map. This matrix has values of 0 for free positions and values of 1 for an occupied position, so it is a binary image.

This function easily finds the circles when the map is accurate. However, if the map is not precise (which is the case when we work with odometry and little GPS), the tables are sometimes represented by a few scattered pixels and therefore not detected by the function.

To overcome this, we did pre-processing on the matrix. First, we make an inflate of it. This allows us to inflate the dot amats. Secondly, we quadruple the resolution of the image. Indeed, by default it is 150×150 pixels, and the tables are therefore a few pixels in radius, which is not easy to detect by MATLAB. Finally, we vary the parameters of the `imfindcircles` function in order to make it more flexible.

When the center of each table is detected, we manually set the radius to $40\,\mathrm{cm}$ (because we know that the radius of the tables is $40\,\mathrm{cm}$ by checking in the simulator). If we didn't have this data, we would simply have taken the radius values returned by MATLAB.

This detection is quite robust; even if the map representation is approximate, tables are still detected (example in Fig. 5).

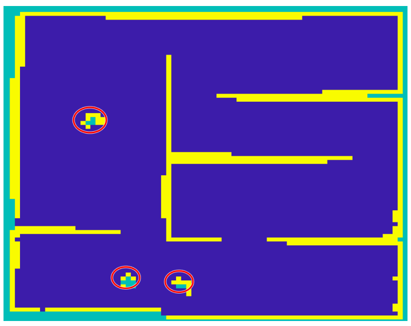


Fig. 5: Table detection on an approximate map.

If MATLAB cannot find all three tables, we restart a search by varying the parameters and making the search more flexible. Our search is therefore based on the assumption that three tables are present on the map and that each table has a radius of $40\,\mathrm{cm}$. This assumption can be modified in the `main.m` file of the project. If MATLAB really does not find the circles, we stop the manipulation and display an error message.

Remark: we have fixed the assumptions concerning the tables because we are certain, given the project statement, that the tables will always be identical. If we hadn't had the assumptions, our detection would still have worked. We would simply have to filter the detections to be sure that they each correspond to a table (by visiting the place and analyzing it with a 3D point cloud for example).

1) Circle Hough transform: The MATLAB's `imfindcircles` function allows us to detect circles in an image via a circle Hough transform. This process is similar to the Hough transform, which we studied during the Computer Vision course, which allows lines to be detected. This function therefore works with the equation of a circle rather than the equation of a line.

B. *Table exploration and analysis*

The second step consists in exploring each of the detected tables and analyzing it in order to determine its type (empty, easy or hard).

So we start by bringing the robot close to a table (we explore the tables from the closest to the furthest away from the robot). To do this, we generate 15 points around the table (with a margin of a few centimeters so that the points are far from the table) and we bring the robot to the point that is closest to it. Then we align the robot with the center of the table. Knowing the position of the robot and the position of the center of the table, it is easy to know the angle to be aligned to the table.

When we don't use GPS (or very little), this alignment step is more complex. Indeed, the center of the table and the position of the robot are both based on estimated data and not exact data. To overcome this problem, we make the robot align towards the center of the table using a 3D point cloud: we filter the points to keep only those of the table and we find the closest point to the robot. Then we align the robot facing this closest point.

Then, when the robot is aligned facing the table, we bring it a few centimeters closer (using distances calculated on the basis of Hokuyo) and we take a 3D cloud point (quite large) of the table. We eliminate the points that are too distant and too low so that we keep only the points representing the objects. We then use MATLAB's `pcsegdist` function to segment the points into different clusters. We set that two points are considered to belong to two different clusters if they are more than $10\,\mathrm{cm}$ apart. Thus, on the empty table we get no clusters, on the easy table we get 5 (or 4 depending on the viewing angle) clusters, and on the difficult table we get 1 cluster. We can then easily determine the type of the table according to the number of clusters found.

An example of the clusters obtained is shown in Fig. 6 and 7.

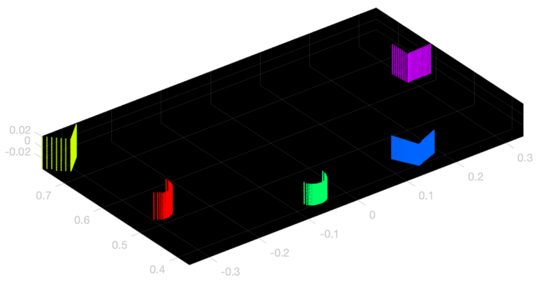


Fig. 6: Easy table clusters.

The advantage of this method is that it is particularly robust: it does not depend on the color or shape of the objects. Initially, we took a photo and analyzed the colors present in the photo. This technique worked, but was color-dependent, which was not robust.

In order to save time for further manipulation, we already calculate the (central) position of the objects on the basis of these clusters. These positions will be used to grab the objects.

For the difficult table, in order to separate the objects into different clusters, we re-use MATLAB's `pcsegdist`

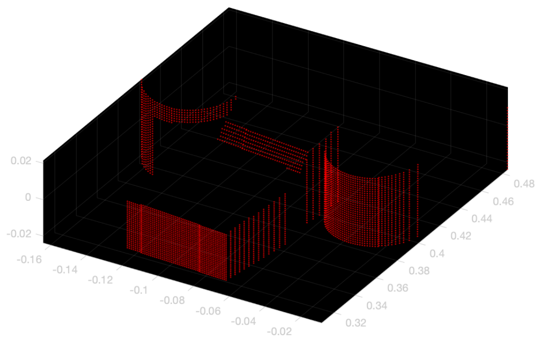


Fig. 7: Hard table cluster.

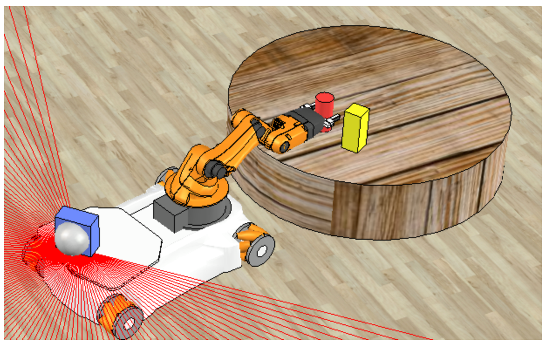


Fig. 8: Fixed grab and drop position of the robot.

function, but this time with a smaller distance between each cluster.

To calculate the position of an object, we isolate all the points in its cluster and simply calculate the average position of all the points. These positions are then converted into the absolute coordinate system. Knowing the position and orientation of the robot, the conversion from relative to absolute coordinates is trivial.

1) `pcsegdist` function: This function of MATLAB allows us to separate the different points of a 3D point cloud from a clustered point. We learned about its operation in order to understand it: the function simply calculates the Euclidean distance from a point to all its neighbors and decides, based on this distance, if the point belongs to the same cluster as its neighbors or if it forms a new cluster. The threshold on the distance is a configurable data.

C. Grabbing an object

The third step is to grab an object. To do this, we start by making the robot go near the object. We use the same system as before: we generate points around the table and take the point closest to the object (the absolute position of the objects was determined in the previous step). When the robot has reached this point, we align it with the object (same principle as for aligning the robot with the center of the table).

Again, when we do not have access to GPS frequently, relying only on the position of the object to align the robot is not always reliable. In this case, we take a 3D cloud point when the robot is close to the table and align it with the center of the table (with the same 3D cloud point analysis as the previous step). The robot is then aligned with the center of the table and not with the object, but it doesn't really matter.

When the robot is globally aligned with its target, it then moves to the fine-tuning phase. Indeed, to grab an object, we use the arm and the gripper always in the same way and at the same position (example in Fig. 8). The robot must therefore be positioned well in front of and at a precise distance from the object ($33\,\text{cm}$).

To align the robot well in front of the object, we take 3D point clouds, isolate the point cluster of the closest object and calculate its central point. According to the x-coordinate of this point, we can tell if the robot is well aligned or not (if $x = 0$, the robot is well in front of the object, otherwise it must slightly turn left or right).

When the robot is perfectly aligned, we must put it at a precise distance from the object. To do this, we take 3D point clouds again and calculate the central point of the closest object. We calculate the distance between this point and the sensor of the robot. According to the found distance, we know if we have to move the robot forward or backward slightly.

Finally, when the robot is well aligned and at the right distance, we make it do a half-turn and grab the object with its arm, always in the same way. When the object is grabbed, we put the arm back to a fixed position so that the object is on the tray, stuck in the gripper.

This technique has the merit of being quite simple and functional. It can however fail because the robot does not necessarily align itself facing one of the faces of the object (if the object is a parallelepiped for example). In this case, the gripper may not grab the object.

This problem occurs very rarely because, even when the robot is not in front of a face, the gripper manages to grab the objects. However, we still implemented a safety mechanism: when the robot grabs (or not) an object, a picture of the gripper is taken. This photo is processed (cropped, reduced and transformed into black and white) and then compared, via a correlation coefficient, to a reference photo of the empty gripper. If the correlation is strong, it means that the gripper is (probably) empty and therefore that the robot has missed its grab. Conversely, if the correlation is low, the grab was successful. When the grab fails, the robot goes back to the position and tries again directly to grab the object.

An example of photos taken and processed is shown in Fig. 9.

This mechanism is quite robust and has been proven several times in our simulations.

D. Dropping an object

The last step is to place the grabbed objects on the objective table.

When the robot has grabbed an object, we start by planning a path to the objective table. To do this, we use the same technique of the previous steps: we generate points around

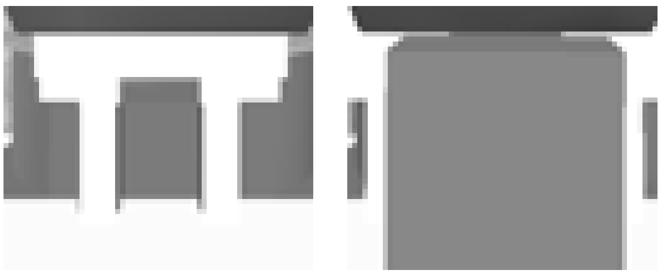

(a) Reference photo. (b) Photo with grasped object.

Fig. 9: Photos taken by the robot to check the grab.

the table and take the closest one to the robot. However, for this step, we try not to take the same one twice so that the robot does not drop two objects in the same place.

When the robot is close to the objective table, we align it with its center. As before, in cases where we have no (or little) access to GPS, we use a 3D point cloud to align the robot.

Once the robot is aligned with the center of the table, we bring it closer to a certain distance (calculated on the basis of the Hokuyo) and make it do a half turn. Finally, he just has to put the object on the table. As for the grab, we use a fixed position of the gripper (see Fig. 8) to drop the objects.

This technique works very well. In very rare cases, the objects have fallen after being placed on the table. We think that this behavior, a little hazardous, comes from the simulator and not really from our technique. Unfortunately, when the object falls (and rolls off the table), there is not much we can do about it.

E. Manipulation finished

When the robot has dropped an object, it returns to the table with objects. If it still has object positions in memory, it will face the object and try to grab it. If, on the contrary, it has grabbed all the objects it has spotted during the analysis of the table, it makes a wide 3D cloud point of the table. If it detects objects again, the process starts again. If it doesn't detect any more objects, the manipulation is finished!

In case of a difficult manipulation, the robot does not stop the manipulation but restarts the whole process with the hard table.

Considering the different verifications made (frequent 3D point cloud and pictures were taken), we are almost certain that the robot will grab all the objects and won't forget any of them. Only the drop of the objects is an operation that we cannot verify.

F. The hard table

We tried to carry out the milestone manipulation not compulsory consisting in transporting all the objects of the hard table.

The main difficulty of this table is that all the objects are gathered in a single heap. So the first step is to try to segment the 3D point cloud representing the heap of objects in order to be able to treat each object individually. For this, we used the same technique as before with a much smaller distance between clusters. We obtained the segmentation shown in Fig. 10.

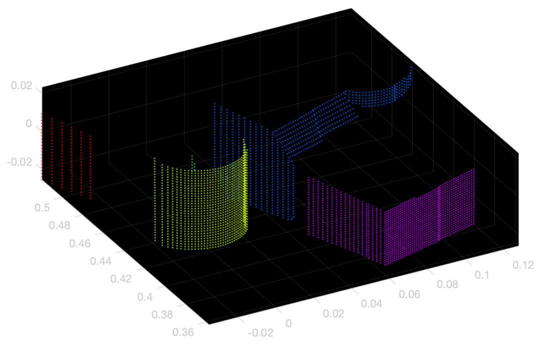


Fig. 10: Segmented 3D point cloud of the hard table.

This segmentation is not perfect but seems to be suitable to detect the different objects.

The second difficulty is that all the objects are very close to each other and therefore grabbing an object usually leads to touching the objects next to it and deflecting the gripper from its objective. For this, we see no other solution than to grab the objects in a very precise order. We have not been able to implement this.

A last difficulty is grabbing the blue lying object. Indeed, given its position, it is not possible to grab it from anywhere. We have to be in a very precise place, depending on how the arm is positioned to grab the object.

By leaving our code as it is, the robot manages, in a few tries, to grab several objects from the hard table but in an inelegant way. It does not manage to grab the last blue object lying down.

A video illustrating our robot on the hard table is available via the following link:

https://youtu.be/FpIr6F7JF6c

The robot manages to grab 4 of the 5 objects and makes 1 fall to the ground.

VI. Retrospective analysis

Our implementation has some positive points that we are proud of, but also some negative points that we could improve in the future.

First of all, we are quite satisfied with our code: it is clean, well organized and commented. The object-oriented paradigm allows for clean use and easy maintainability. We are also happy with its robustness: problem situations are, in a majority of cases, detected and handled correctly.

Secondly, we are also happy with the robustness of some of our algorithms: the detection of tables on the map is really reliable, the segmentation of objects in 3D point clouds works quite well and the object grab check mechanism avoids empty round trips between tables.

Concerning the object grasping, our mechanism is not very accurate. Despite our verification system, the robot sometimes grabs objects in an inelegant way. This could have been improved by making a more thorough analysis of the 3D point cloud to determine the position of the faces of the objects and thus be able to align perfectly in front of them.

We are quite happy with our odometry. This seems to work relatively well: we get a representation of the environment that is almost as precise as with GPS. However, in rare cases the representation is poor (some pieces of the wall are forgotten since the last Hokuyo ray to see them was parallel to that wall, etc.). We haven't found a robust way to avoid this. But being quite rare cases, we are not too worried.

Finally, regarding pure SLAM, we are disappointed that we did not get there. We tried quite a few techniques, but each time was unsuccessful. We realized that it is not an easy task and that it takes more time than we expected.

VII. Conclusion

This project allowed us to explore a part of the robotics world. Despite the simplifying assumptions, the simple environment and the fairly easy tasks to perform, we realized that it is not easy to create an autonomous robot.

We have learned to use route planning algorithms as well as sensors to discover and analyze an environment as well as to construct a relevant representation of it.

We also learned to manipulate more mechanical aspects of the robot: use of its arm and gripper, precision placement, etc. We made him do simple tasks like grabbing and carrying objects on his own.

All of these tasks have been implemented and completed successfully. Using GPS leads to great results. The use of odometric data leads to less precise results, but quite acceptable and allowing the robot to perform these tasks.

However, despite the very large number of hours spent on this project; far exceeding the course workload assessment, we were unable to implement even one of the two non-mandatory milestones; we are a little disappointed by this.

Autonomous control of a robot in a real (simulated) environment is far from easy given the unknown situations and environment. This course allowed us to have the first glimpse of its difficulties and to open the door to the large world of autonomous robots and robotics in general!

References

[1] D. F. Sebastian THRUN, Wolfram BURGARD, *PROBABILISTIC ROBOTICS*. Ronald C. Arkin, 2000.

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

[3] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. D. Reid, and J. J. Leonard, "Simultaneous localization and mapping: Present, future, and the robust-perception age," *CoRR*, vol. abs/1606.05830, 2016. [Online]. Available: http://arxiv.org/abs/1606.05830

[4] Renaud Detry, "Trs: An open-source recipe for teaching/learning robotics with a simulator," 2014. [Online]. Available: http://ulgrobotics.github.io/trs/

[5] ——, "Trs: An open-source recipe for teaching/learning robotics with a simulator," 2020. [Online]. Available: https://github.com/ULgRobotics/trs

[6] Coppelia Robotics, "Regular api function list (alphabetical order)," 2020. [Online]. Available: https://www.coppeliarobotics.com/helpFiles/en/apiFunctionListAlphabetical.htm

[7] Cyrill Stachniss, "Slam-course - 01 - introduction to robot mapping (2013/14; cyrill stachniss)," 2013. [Online]. Available: https://www.youtube.com/watch?v=wVsfCnyt5jA

[8] MATLAB, "imfindcircles - find circles using circular hough transform," 2020. [Online]. Available: https://nl.mathworks.com/help/images/ref/imfindcircles.html

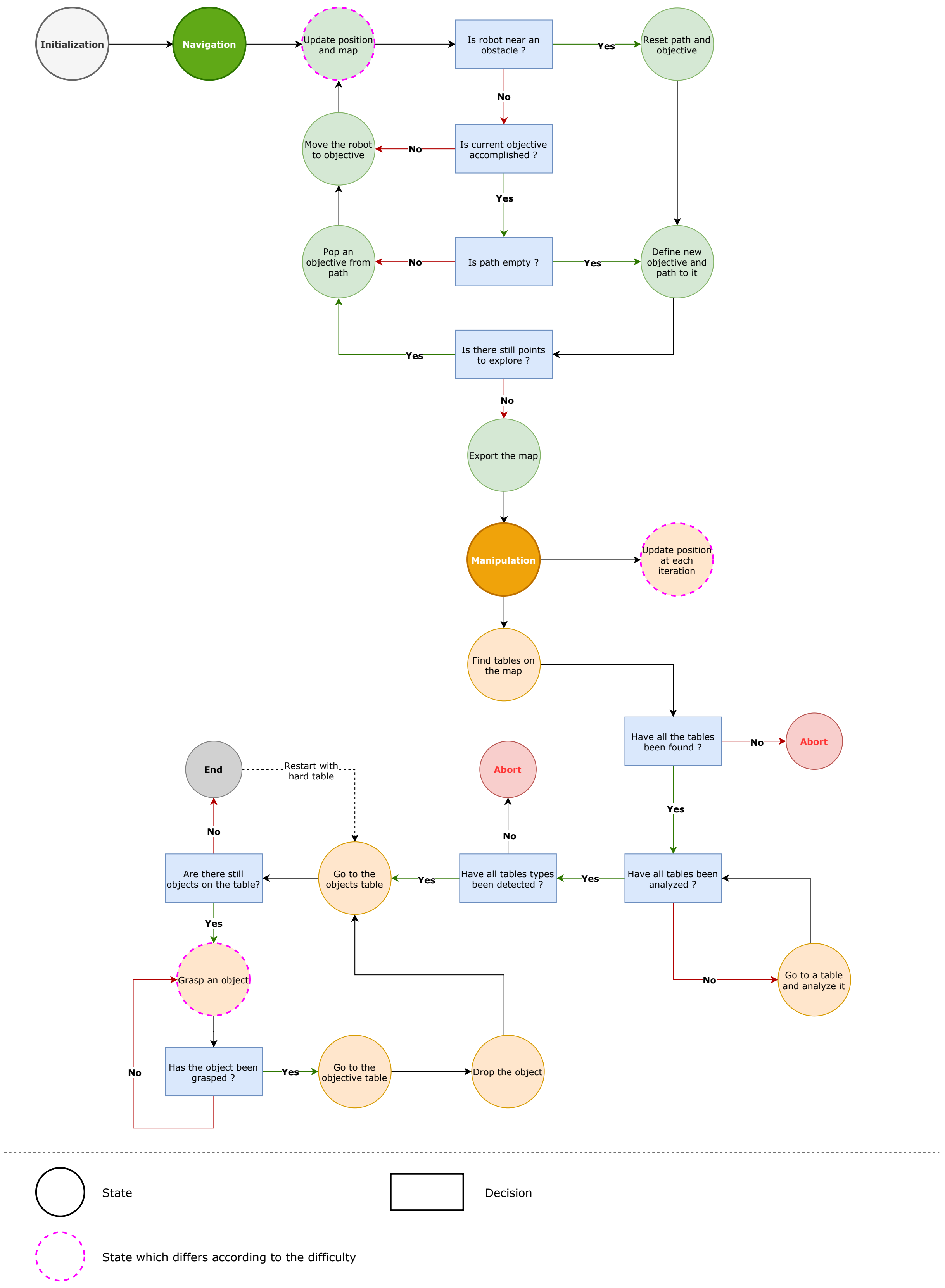
[9] ——, "pcsegdist - segment point cloud into clusters based on euclidean distance," 2020. [Online]. Available: https://nl.mathworks.com/help/vision/ref/pcsegdist.html

[10] ——, "What is slam?" 2020. [Online]. Available: https://nl.mathworks.com/discovery/slam.html

[11] Wikipédia, "Circle hough transform," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Circle_Hough_Transform

[12] ——, "Simultaneous localization and mapping," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping

[13] Scott Martin, "What is simultaneous localization and mapping?" 2019. [Online]. Available: https://cutt.ly/IfdwyAN



Initialization
Navigation
Update position and map
Is robot near an obstacle ?
Yes
Reset path and objective
No
Move the robot to objective
No
Is current objective accomplished ?
Yes
Pop an objective from path
No
Is path empty ?
Yes
Define new objective and path to it
Yes
Is there still points to explore ?
No
Export the map
Manipulation
Update position at each iteration
Find tables on the map
Have all the tables been found ?
No
Abort
Yes
End
Restart with hard table
Abort
No
Are there still objects on the table?
Go to the objects table
Yes
Have all tables types been detected ?
Yes
Have all tables been analyzed ?
No
Go to a table and analyze it
Yes
Grasp an object
No
Has the object been grasped ?
Yes
Go to the objective table
Drop the object
State
Decision
State which differs according to the difficulty

Fig. 11: Complete finite state machine of the youBot.