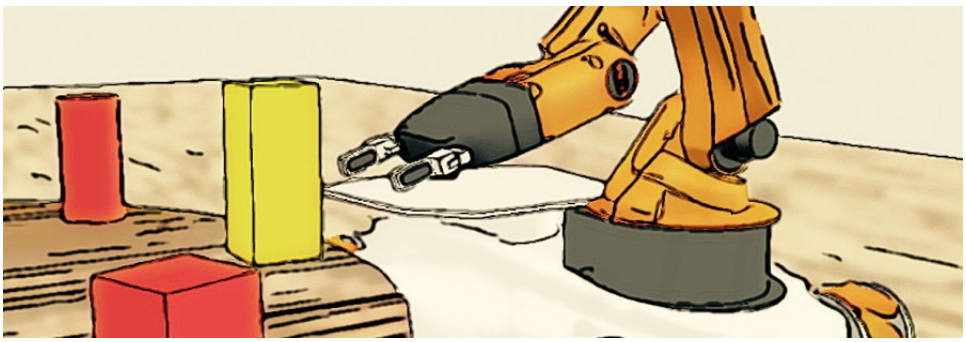
University of Liège

Academic year 2018-2019

# Milestone A1
# Introduction to intelligent robotics



Arthur Robert - Mohamed El Osrouti
MASTER 1 Aerospace engineering - Electrical engineering

# Introduction

Video link: INFO0948 - Final Youbot V1 A. ROBERT/M. EL OSROUTI
The *Introduction to intelligent robotics* project is about developing the youbot to make it execute some tasks in its environment. Following a list of the main subjects of the project :

- Navigate in our environment while mapping
- Interpret youbot position and orientation
- Manipulate objects from the tables
- Find and identify each basket in the house
- Combine manipulation and vision to put objects in the right basket

In this project we assume the map is unknown and of an unknown size. Our initial position regarding the whole map is also unknown.

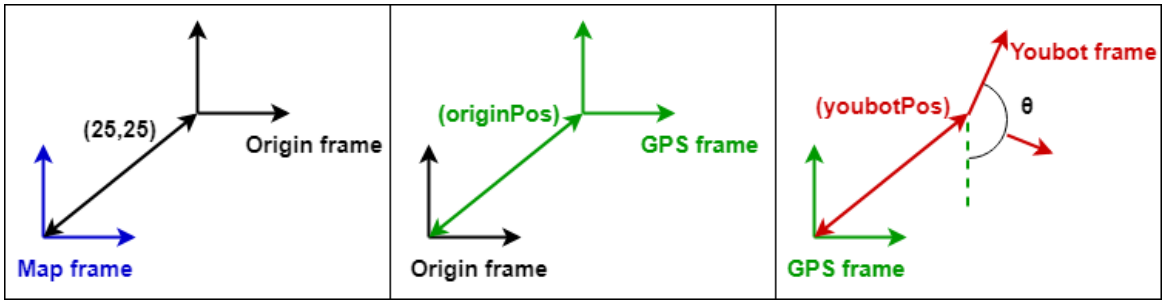



Figure 1: Project main frames

To make it easier to use we have defined 4 frames (see Figure 1) :

- **Map frame** : The map frame is defined as the bottom-left of the occupancy grid
- **Origin frame** : Located at the center of the occupancy grid, it represents where youbot is starting regarding the map
- **GPS frame** : Coordinates given by youbot's GPS
- **Youbot frame** : Youbot relative frame. Note : youbot z angle is defined as 0 on the $-\vec{y}$ axis

In the end, we can define some frame transfer functions :

| | |
|---|---|
| GPS $\rightarrow$ *map* | $(x,y)_{map} = (x,y)_{GPS} - originPos + size(map)/2$ |
| map $\rightarrow$ *GPS* | $(x,y)_{GPS} = (x,y)_{map} + originPos - size(map)/2$ |
| youbot $\rightarrow$ *origin* | $(x,y)_{origin} = (x,y)_{youbot} - originPos$ |
| origin $\rightarrow$ *youbot* | $(x,y)_{youbot} = (x,y)_{origin} + originPos$ |
| youbot $\rightarrow$ *GPS* | $(x,y)_{GPS} = rotationMatrix(\theta) * (x,y)_{youbot} + youbotPos$ |
| GPS $\rightarrow$ *youbot* | $(x,y)_{youbot} = rotationMatrix(\theta)^{-1} * ((x,y)_{GPS} - youbotPos$ |

To achieve mapping and navigating inside the house we mainly use the MATLAB's *Robotics System Toolbox*.

# Mapping

As said in the introduction the map is considered of unknown size. Plus, our initial position regarding the whole map is also unknown. A solution would be to put youbot in a random position in our map and then move rows and columns if needed. But this solution would take too much computation time. So we decided to create a map big enough to contain the whole house even if we don't know where we are at the beginning. So we define a map, 50x50m with a resolution of 8 cells per meter.

To map we use the **OccupancyGrid** class. It provides a 2D occupancy grid map based on our the width and heigth we want and a resolution (number of cells par meters). Each cell has a value representing the probability of its occupancy. A value close to 0 is a free cell, close to 1 is occupied and around 0.5 is unknown (see Figure 2). **OccupancyGrid** is also able to inflate occupied cells for obstacles avoidance.

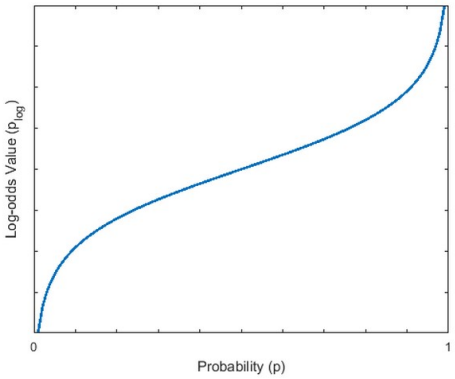



Figure 2: OccupancyGrid probability

## Sensor information

Youbot use a Hokuyo sensor to give information about it's vision. They are expressed in youbot's frame so to update the **OccupancyGrid** we need to take into account youbot has an $\theta$ angle between it's front axis and GPS $-\vec{y}$ axis. We receive a 3*N (x, y, z) matrix containing Hokuyo rays end points ans a other 1*N logical matrix if an end point is on a obstacle.
We now need to express these points in the map reference to update our **OccupancyGrid**.

$$X_{map} = X_{youbot} * \cos(\theta) - Y_{youbot} * \sin(\theta) + youbotPos - originPos + size(map)/2$$

$$Y_{map} = X_{youbot} * \sin(\theta) + Y_{youbot} * \cos(\theta) + youbotPos - originPos + size(map)/2$$

So it comes :

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}_{map} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}_{youbot} + youbotPos - originPos + size(map)/2$$

## Occupancy grid

So we have the information from our Hokuyo sensor, but it's not enough. We have information on the end of laser rays but we don't have information about what's happening between (see Figure 3). In the beginning we used **inpolygon** function to return intermediate points between youbot and known points. But this function needed a meshgrid based on the basic house size and was very costly. So, in the end, we decided to use **OccupancyGrid.insertRay** which needs only our youbot position and the position of points we know (see Figure 4). It will automatically compute those intermediate points and update their occupancy probability based on mid and

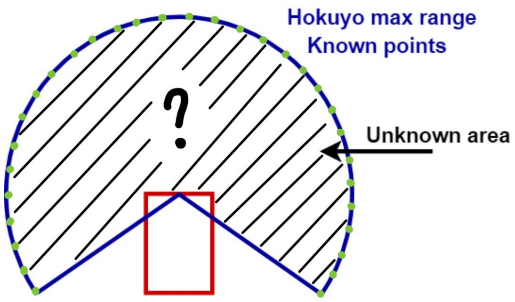



Figure 3: Hokuyo sensor

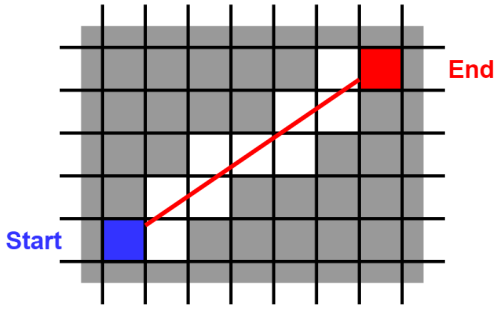



Figure 4: Hokuyo sensor

end points probabilities we set when calling it.
Plus we use **OccupancyGrid.updateOccupancy** for obstacles. Because Hokuyo has limited range, returned information is not necessarily a obstacle.
Hokuyo provides a large number of point so we decided to downsample our list. It's not a problem because if we have a too large number of ray casting they will update the same cells. Plus we compensate the number of ray casting with the large number of map updates.

## Mapping end

We now need a condition to stop mapping. We decided to look at our map occupancy. **OccupancyGrid.occupancyMap(map, 'ternary')** allows us to get our map matrix and the term **ternary** makes it easier to compute how many unknown points are inside the map (because returns -1 for unknown, 0 for free and 1 for obstacles). If this value doesn't change when we reach a path 4-5 times in a row we assume youbot is not discovering anymore and mapping is over.

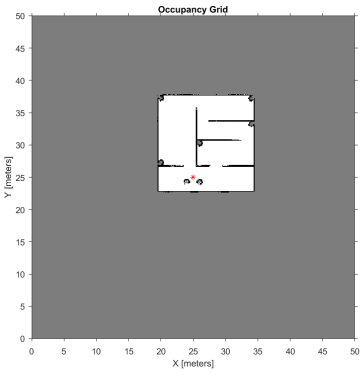



Figure 5: Mapped **OccupancyGrid**

# Navigation

First thing in our navigation is we use an inflated copy of our map to avoid using **OccupancyGrid.inflate(map, radius)**. Then we try to choose a interesting point to navigate to using different algorithms. We then plan a path using **PRM (ProbabilityRoadMap** class which places random points on our map and connect them in relation the maximum distance we pass as parameter.

- We retrieve position information from points we already visited. If they are close enough, less than 5-6 meters, we compute the angle between youbot's position and these points. We then do a 'mean' value using custom weight. Closer they are, heavier the point will be in the computation. We then use the **meanAngle(angles)** which is basically the following formula :

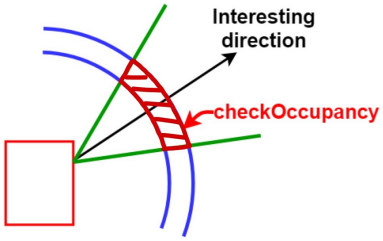


Figure 6: Check occupancy

$$atan2(\frac{1}{\sum wgt} * \sum wgt_i * \sin\theta_i, \frac{1}{\sum wgt} * \sum wgt_i * \cos\theta_i)$$

- We meshgrid around youbot's position then compute each point norm and angle regarding to $-\vec{y}$ axis. We retrive only points between 3 to 4 meters and if the angle is in a cone of $meanAngle \pm 15°$ using (see Figurre 6) :

```
R = hypot(x, y);
T = atan2(x, -y) - meanAngle(agls, wgt);

cells = [x(4 > R > 3 & abs(T) < 2*front_angle)...
         y(4 > R > 3 & abs(T) < 2*front_angle)]...
         + youbotPos_map(1:2);
```

  We then retrieve matrix indexes from these coordinates to remove duplicates, get free cells and choose one randomly. Now transform the target indexes to coordinates to plan a path. **If no free cells was found we increase the plus/minus angle to have a wider cone until we finally find one.**

- We update the **PRM** to take into account what we discovered since last update. We already have the target to reach so we just call **PRM.findpath(prm, start, end)** which will return a vector composed of our start, our goal and some intermediary points. We also push these points inside a **FIFO** list called **Queue** used for driving.

# Driving

To drive youbot we are mainly using **PurePursuit** which will do most of the computation. Youbot has omnidirectionnal wheels, so we still need to transform data we receive from **PurePursuit** to make it move sideways, forward and rotating at the same time. As said previously, we use a **FIFO** list to keep a track of our current intermediate target. Each time we reach one of these targets we $popfront()$ and set the next value as current target.
**PurePursuit** needs our position plus our orientation to return a forward and a rotation speed. In fact, **PurePursuit** and our frame don't have the same angle reference. So we need to add $\pi/2$ to make it works correctly.

Now we have our **PurePursuit** driving our youbot but it doesn't use the specificity of omnidirectional wheels. So we need to break down our front speed in front and side speed. So we already know youbot's angle ($\alpha + \beta = \theta$) as **youbotEuler** so we just need to compute $\beta$ to get $\alpha = \beta - \theta$ (because in this case $\alpha$ in negative). We then use almost the same rotation matrix as used in sensor information conversion.

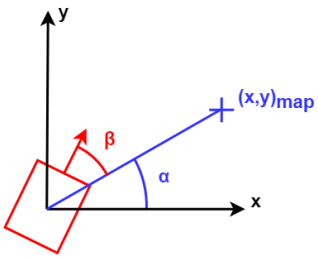



Figure 7: Driving angle

# Vision

## Baskets/Tables detection

After the mapping process, we get an occupancy map made of free cells (0 values) and obstacles (1 values). The next objective is to localize the baskets and the tables in the map. There are mainly two ways to identify them :

1. **Online mode** : during the mapping and navigation, the youbot tries to catch curved signals from the Hokuyo sensor.

2. **Offline mode** : after mapping, the robot tries to find circles or curved segments in the map (since some baskets are located at corners) by using the function **imfindcircles** from matlab.

We chose the second solution since it allows the robot to focus only on the mapping-navigation process without adding any delay. However, the resolution should be high enough in order to identify without ambiguity the tables and baskets. It's the reason why we increased the map resolution from 8 to 16.

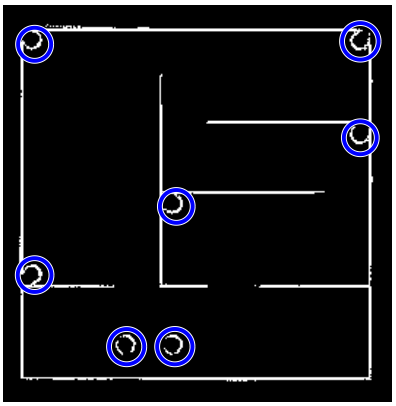


Figure 8: Localization of the tables and baskets thanks to **imfindcircles**.

As shown in Figure 8, the tables and baskets are pretty well localized by their centers and radius. Since there are 7 tables and baskets, we take the 7 more relevant circles found on the map.

## Identifying baskets

We know where are the baskets but we don't know yet which object is associated to which basket. To link an object to a given basket, after multiple attempts, we decided to use the **SURF (Speeded Up Robust Features)** to retrieve a bag features from the given pictures. As specified in the **Computer Vision System Toolbox** (see Figure 9), the most adapted algorithms to our problem seems to be the SURF since it's scale and rotation invariant and can be used for image classification.

| Descriptor | Binary | Function and Method | Invariance | | Typical Use | |
|---|---|---|---|---|---|---|
| | | | Scale | Rotation | Finding Point Correspondences | Classification |
| HOG | No | `extractHOGFeatures(I, ...)` | No | No | No | Yes |
| LBP | No | `extractLBPFeatures(I, ...)` | No | Yes | No | Yes |
| SURF | No | `extractFeatures(I, 'Method','SURF')` | Yes | Yes | Yes | Yes |
| FREAK | Yes | `extractFeatures(I, 'Method','FREAK')` | Yes | Yes | Yes | No |
| BRISK | Yes | `extractFeatures(I, 'Method','BRISK')` | Yes | Yes | Yes | No |
| • Block<br>• Simple pixel neighborhood around a keypoint | No | `extractFeatures(I, 'Method','Block')` | No | No | Yes | Yes |

Figure 9: Feature descriptors from **Computer Vision System Toolbox**

We then compare this bag of features with the features from the photo we take with youbot. As shown in Figure 10, the SURF algorithm is able to recognize the dog from the left picture on the right picture. Even if its orientation or position is different.

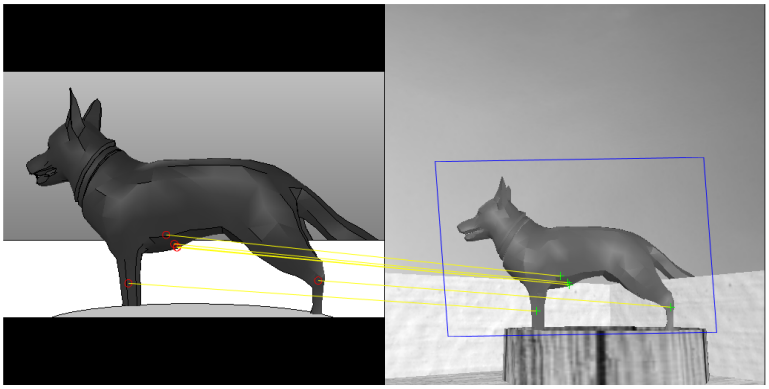


Figure 10: Features match from given picture (left) and photo taken by youbot (right) + position of the dog indicated by the blue quadrangle

# Conclusion

We finally managed, with a (almost) complete rework of our work after the mid-project review, to map the house with great performances. Computation time was really improved when removing **inpolygon**. However we still, sometimes, have some issues with the navigation. The choice of the next point to discover may require additional conditions/modifications to have a very reliable algorithm. We still manage to map the whole house with a rate of around 80
Concerning driving, youbot is able to drive in all directions using its omnidirectional wheels.
To identify tables and baskets we had to methods. We chose to detect baskets after mapping is ended because we though it was easier.
In the end, our code seems to provide good results even if we may need to rework some parts to become fully reliable.