Turtlebot using ROS Robotics Project

Laboratory sessions and project report

KIRILL FROLOV - LUIS MIGUEL ZAPATA

Université de Bourgogne - 2015

Contents

1 Introduction

This document presents the work done in the laboratory sessions as well as the final project corresponding to the subject: *Robotics Project* coursed in the *Master in Computer Vision* program of *L'Université de Bourgogne*. These sessions concern the use and implementation of algorithms related to *Mobile Robotics* using the *TurtleBot* and the *ROS (Robotics Operating System)*.

1.1 Mobile Robotics

A *Mobile Robot* is any kind of device which is able move in its environment due to the use of actuators and possibly, but not necessarily, having some sensors to obtain information about its environment. For environment it is understood the media in which the Robot makes displacements, it can be air, water, some surface or why not, the vacuum from outer space. The Mobile Robots are defined as *Service Robots* and their purpose is to achieve some goal, for instance, to go from point a to b.

1.2 Turtlebot

The *TurtleBot*, which is the robot that will be used during the whole subject, is constructed as a *Differential Drive Robot*, it has two driving wheels connected to motors and two support wheels to give stability. Such configuration allows the robot to move in indoor environments with flat surfaces such as the Robotics Laboratory floor. The TurtleBot is equipped whit several sensors to be able to navigate autonomously, between those sensors there can be found encoders connected to each driving wheel that allow to calculate the displacement of the robot. Besides this, there is one *RGB-D Camera (Kinect)* and a Laser Scanner *LiDAR* provided by the laboratory staff that not comes originally with the TurtleBot.

Figure 1: Turtlebot

1.2.1 Kobuki base

The *Kabuki Base* is the base controller of the differential drive system and therefore is in charge of moving the robot. This system was integrated thanks to ROS to other sensors using a laptop so the robot can be used to do open sorce development in robotics.

1.2.2 Kinect

As stated before the *Kinect* is an *RGB-D* camera that streams color images as well as depth images thanks to its stereo-vision system. The kinect is a really inexpensive device suited to perform *Computer Vision* algorithms in real time thanks to ROS and libraries such as OpenCV and PCL.

Figure 2: Kinect

1.2.3 LiDAR

The *LiDAR* is a laser based depth sensor that rotates 360 degrees and it is perfect for autonomous navigation. This sensor provides information about the surrounding objects in all direction to avoid collisions.

Figure 3: LiDAR

1.3 ROS

ROS which stands for *Robotics Operating System* is an Open Source set of tools developed to do rapid prototyping as well as industrial applications in robotics. It is ideal for doing research and test different algorithms in short amounts of time thanks to its open libraries, once again avoiding to do the implementation of the whole system and as they say it: "Stop reinventing the wheel!".

ROS was developed in order to integrate the different components in a centralized way, for doing so, ROS posses its own *Filesystem* and *Architecture* and the different libraries and components are known as packages.

1.3.1 Filesystem

ROS is only available for Linux platforms and therefore it could be difficult to find the different components which is spreaded in the Linux Filesystem. Therefore ROS has embedded its own *Filesystem*, allowing to perform different task using its own commands and arguments. For example using the command *CD* with *ROSCD (ROS current directory)*

1.3.2 Packages

Packages in ROS are the way the software is organized, normally these packages are composed of libraries, scripts and have dependencies of other packages. To create, modify or visualize the different packages there are several commands based in *ROSPACK (ROS packages)*.

1.3.3 Architecture

The architecture in ROS is based in nodes which communicate with each other trough a centralize node, know as the *ROSCORE*. The core establishes the rules, this is the way the nodes communicate and the conventions to treat the information. In ROS the topology of *Publisher* and *Subscriber* is commonly used and it is done through pre-established *ROSMSG (ROS Messages)*.

1.3.4 RBX Packages

Most of this curse is based either in Goebel (2013) and Goebel (2014) as well as the beginner level tutorials in Various (2015) and of course the lectures of the subject Seulim (2015). The book *ROS by Example I* and *ROS by Example II* present a set of packages that will be constantly used for doing the different tasks presented in the next sections. With them, a lot of components and behaviors can be tested in the Turtlebot.

1.3.5 RBX Packages

These packages contain simulated robots for *RVIZ* which is a 3D Visualizer for sensor data information and the status of the robot. RVIZ will be extensively used to test things before doing it in the real robot and the basic interface can be seen in figure 4.

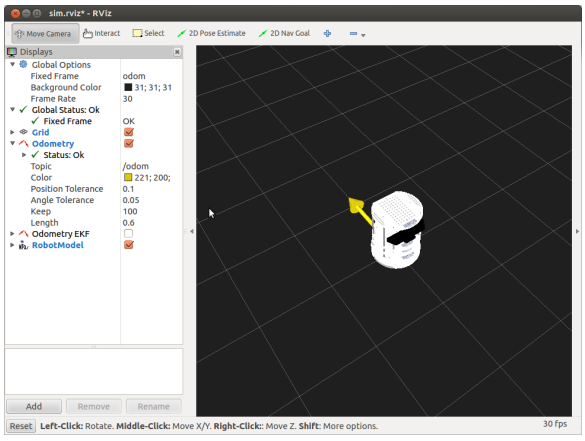


Figure 4: RVIZ (ROS Visualizer)

2 Motion Control

Motion Control refers to the capability to give commands to the Robot in order for it to move. The Motion Control itself is a really complex task divided into several layers that represent different levels of abstraction for the user. These levels are:

- `Motors, Wheels, and Encoders` - These are the physical components to be controlled and sensed.
- `Motor Controllers and Drivers` - In this level the idea is to ensure the desired movement of the robot by controlling the speed of each wheel thanks to the motor and the encoders.
- `The ROS Base Controller` - Is the link between the Robot and ROS, it takes the information provided by the odometer (\odom) and gives commands to the motors velocities using a *PID* controller (\cmd_vel). At this point the commands are described in physical representations such as $\frac{m}{s}$.
- `Move_Base ROS Package` - This package allows to perform motion only by indication the desired goal based in a referenced position, for this the package used *Path Planning* based in cost maps.
- `AMCL and SLAM` - This level allow to perform autonomous navigation either the prior information of the map is know with *AMCL*, that stands for Advanced Monte-Carlo localization or not by using *SLAM* (Simultaneous Localization and Mapping).
- `Semantic Goals` - The higher levels correspond to giving orders to the robot by commands recognized by humans as for example: "Go to the kitchen".

In this sections several of the levels are tested showing how to control the Turtlebot in each of them.

2.1 Twist Messages

As stated before ROS has determined protocols that have been developed to communicate between devices. A very important kind of message is the *Twist Message*, since this is the kind of message used to give velocities commands to the robot. The architecture of this kind of message is as follows:

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Listing 1: Twist Message

Just by publishing into the topic $\backslash cmd_vel$ using the Twist Message architecture the robot can be moved.

```
workstation:
$ roslaunch rbx1_bringup fake_turtlebot.launch
$rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.25, y: 0, z:
    0}, angular: {x: 0, y: 0, z: 0.5}}'
rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

Listing 2: Publishing Twist Message

The first line correspond to bring the fake Turtlebot to be manipulated. The message published commands the fake robot to move in a linear velocity in $x = 0.25\frac{m}{s}$ and in angular velocity of $\theta = 0.5\frac{rad}{s}$ every tenth of second. The last command opens RVIZ to visualize the trajectory of the fake robot.

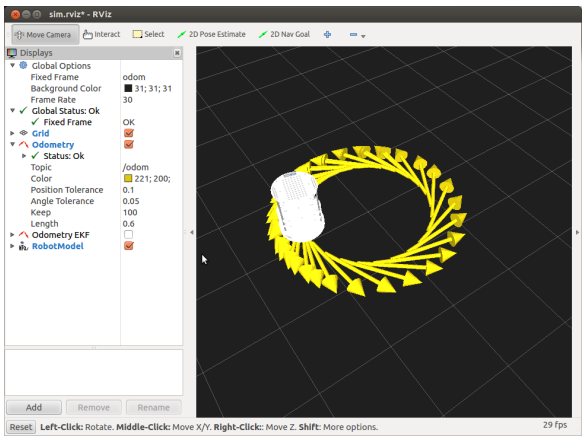


Figure 5: Publishing Twist Message in Fake Turtlebot

Publishing Twist Messages from a ROS Node

The idea here is to run a node that will be publishing onto the ROS topic $\backslash cmd_vel$ constantly so it moves the robot for the desired amount of time. The robot moves totally straight for a period of time, rotates 180 degrees and move straight again to reach its initial point, finally rotates to obtain its initial heading.

```
workstation:
$ roslaunch rbx1_bringup fake_turtlebot.launch
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
$ rosrun rbx1_nav timed_out_and_back.py
```

Listing 3: Timed Out and Back Fake Turtlebot

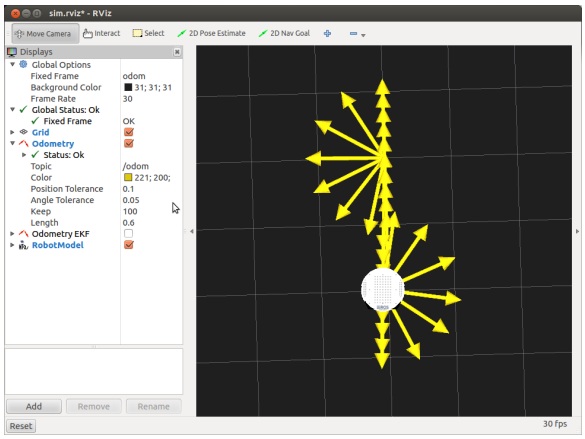


Figure 6: Timed Out and Back in Fake Turtlebot

As it can be seen the robot performs an ideal trajectory, but this was not the case running the same script with the real robot due that the robot has inertia and the controllers can not get to the desired set points immediately.

Publishing Twist Messages and Subscribing to Odom Messages

To overcome this problem in the real robot, the odometer data combined with the gyroscope data is used rather than trusting in the timing constrains using an *Extended Kalman Filter*. The result are far better than before even thought in simulation they appear to be really similar.

```
workstation:
$ roslaunch rbx1_bringup fake_turtlebot.launch
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

```
$ rosrun rbx1_nav odom_out_and_back.py
```

Listing 4: Odom Out and Back Fake Turtlebot

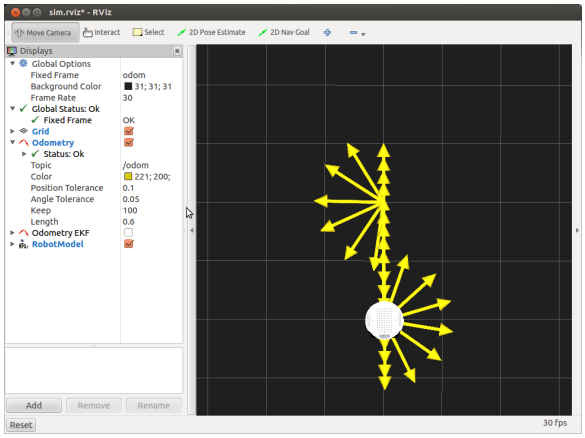


Figure 7: Odometer Out and Back in Fake Turtlebot

Moreover the robot can be moved to different positions, in this case drawing one square and showing that it is not sufficient to use just the combined data of the gyroscope and odometer. The steps to launch the (nav_square.py) script in the fake Turtlebot are shown in the listing 5 and the results can be seen in figure 8, even thought it is an ideal case.

```
workstation:
$ roslaunch rbx1_bringup fake_turtlebot.launch
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
$ rosrun rbx1_nav nav_square.py
```

Listing 5: Odom Out and Back Fake Turtlebot

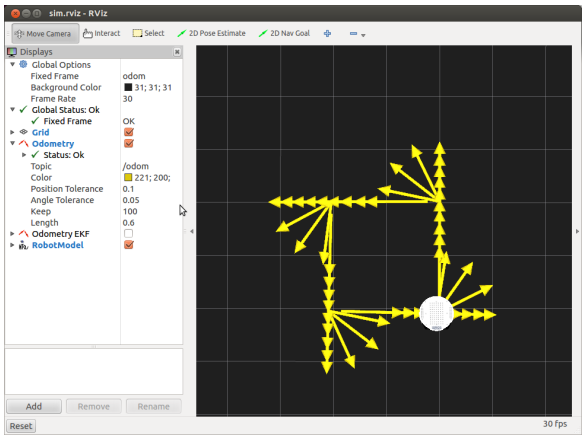


Figure 8: Navigate in a square.

2.2 Move Base

Move base is a better, but at the same time more complex way of moving the robot. The package performs path planning with cost function thanks to a previously provided map to calculate a path and avoid possible obstacles. Different from the Twist messages the (\$move_base\$) messages are based in positions and not to velocities and are also referenced to a certain frame, in this case to the map frame since it is fixed.

```
workstation:
$ roslaunch rbx1_bringup fake_turtlebot.launch
$ roslaunch rbx1_nav fake_move_base_blank_map.launch
```

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav.rviz
```

Listing 6: Odom Out and Back Fake Turtlebot

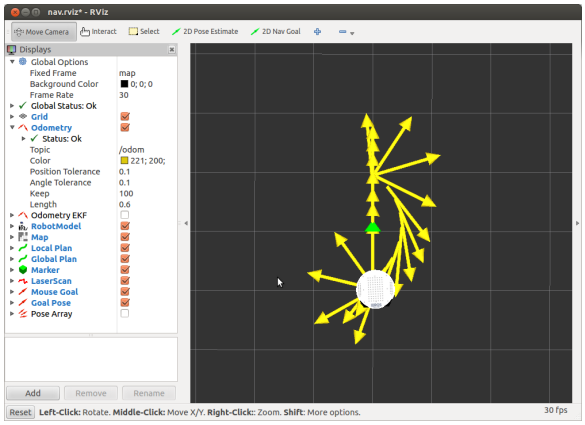
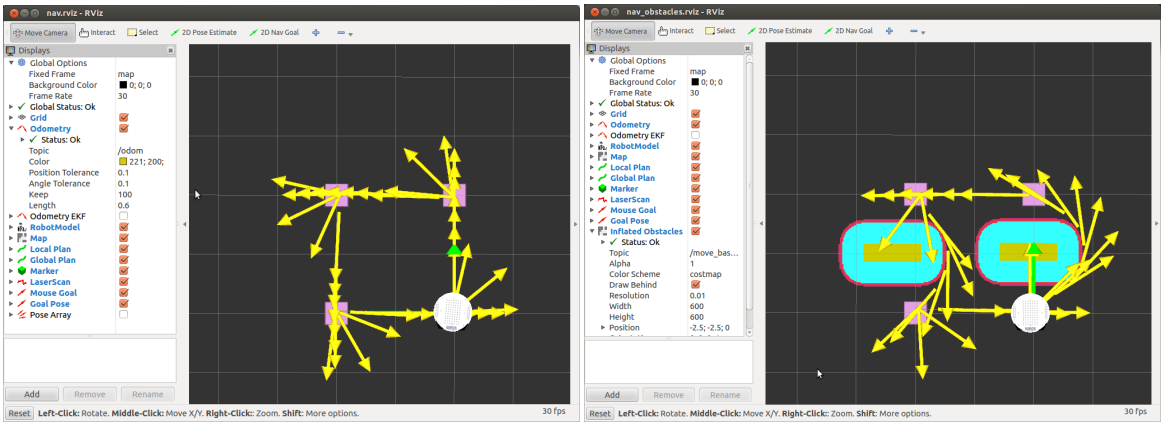


Figure 9: Out and back using comands to move base.

Path Planning and Obstacle Avoidance

Similar as done previously with Twist messages, several movements can be performed using a script node to do a square shape. For so the *rosrun rbx1_nav move_base_square.py* is run. Results for the fake Turtlebot can be seen in figure 10. Regarding the real turtlebot the starting position differs from the final goal.



(a) (b)

Figure 10: Square with move base. (a) Move square, (b) Square with obstacles.

Using the Kinect depth camera it is possible to emulate a laser sensor and therefore obstacles can be avoided in real environments.

3 Navigation and Localization

Along with the common Turtlebot sensors a *Laser Imaging Detection and Ranging (LIDAR)* have been implemented in the turtlebot. For using it the package developed by the *Le2i* is compiled and tested. The package can be cloned from the repository located in `https://github.com/roboticslab-fr/rplidar-turtlebot2`.

The LiDAR emits a modulated Laser pulse which is reflected by the surrounding objects and therefore detected computing the distance to objects up to 6 meters of distance.

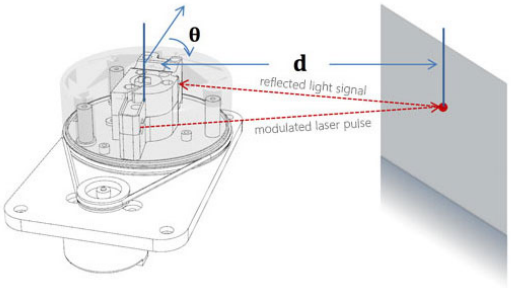


Figure 11: LiDAR internal components

3.1 Map Building using the gmapping Package

A maze was constructed between the Master in Computer Vision colleagues in order to test behaviors of the robot with all the different algorithms in a controlled environment. In figure 12 can be seen the maze built using wood panels.

Figure 12: Maze.

To perform the mapping different scripts are run and can be seen in listing 7 :

```
netbook:
$ roslaunch turtlebot_le2i rplidar_minimal.launch
$ roslaunch rbx1_nav rplidar_gmapping_demo.launch
workstation:
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
$ roslaunch turtlebot_teleop keyboard_teleop.launch
$ rosrun map_server map_saver -f my_map
```

Listing 7: Mapping instructions

The process consists of moving the robot using the tele-operation keys along the maze and generate the map while it moves. As the laser sensor has a range up to six meters approximately most of the maze can be seen from first sight, however first approach made was moving the robot several times around the maze, the result can be seen in figure 13, it can be seen that there are several inaccuracies and that as the mapping uses the odometer for measuring distances the map contains redundant information.

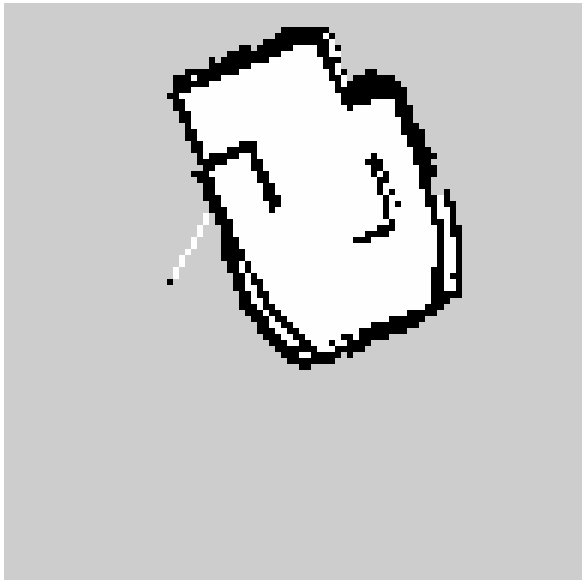
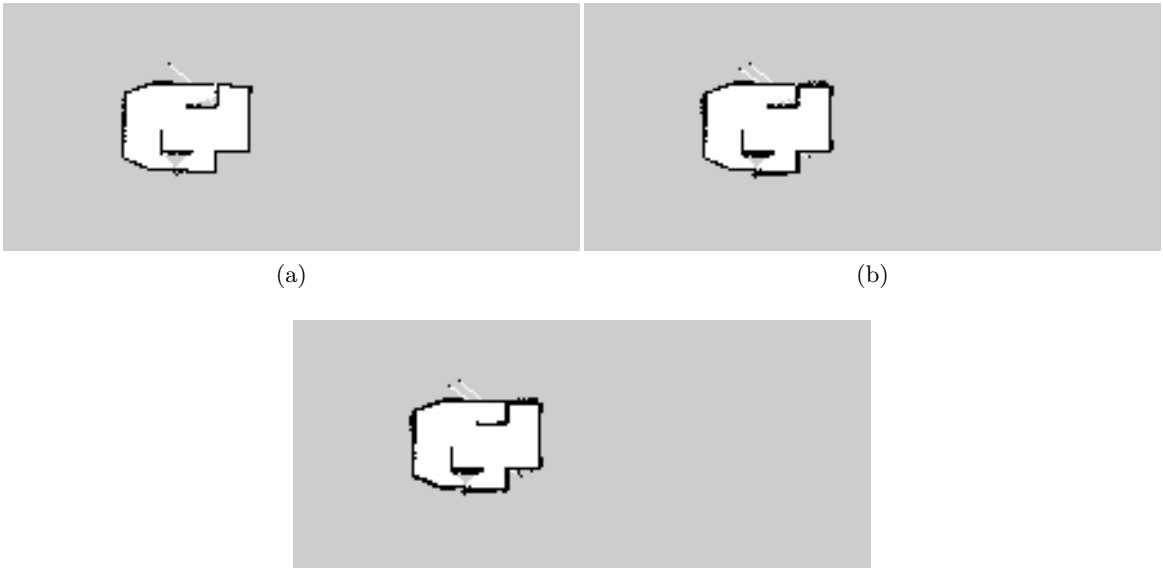


Figure 13: First attempt of constructing the map.

The subsequent intents for building the map were done in just one round but saving results at different steps of time. In figure 14 can be noticed that most of the map is acquired correctly only by moving the robot in an straight line from the left-most to the right-most part of the maze, then in (b) and (c) give place to more imprecision since the odometer accumulates error with time.



(a) (b)

(c)

Figure 14: Map building. (a) First saved, (b) Second save, (c) Third save.

Finally the first saved map is processed and gives birth to the final map seen in figure 15.

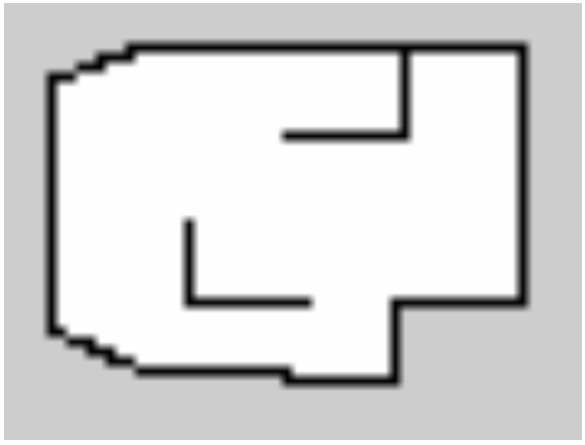


Figure 15: Map constructed with GMapping and post processed manually.

3.2 Navigating in the Real Maze

Similar to the simulation the robot can be easily controlled in the real maze using the RVIZ tools, the map, the current position of the robot and even the AMCL particles can be seen in the interface. See figure 16.

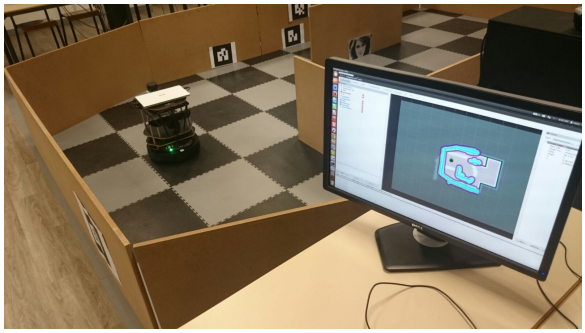


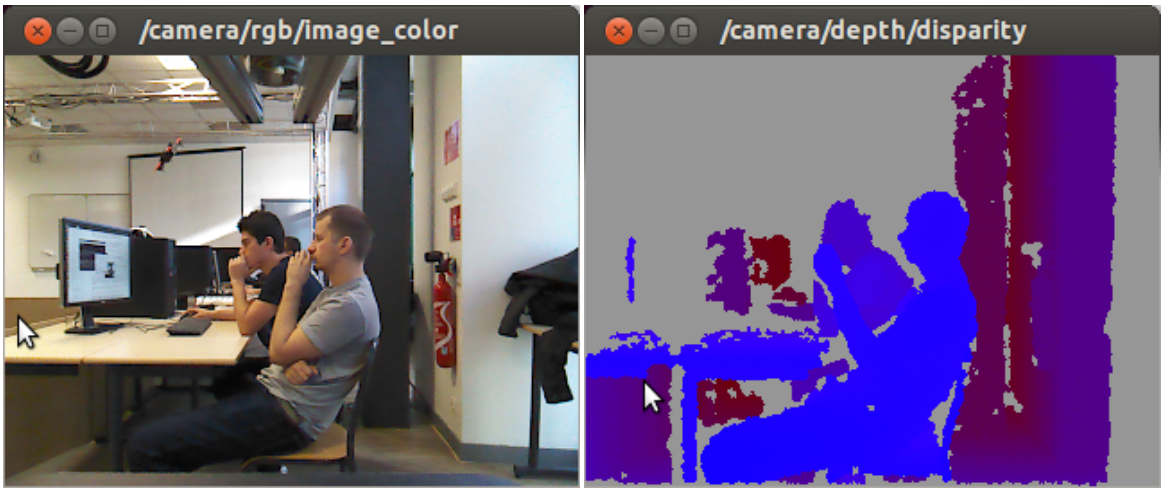
Figure 16: Navigating in the map using real robot.

4 Computer Vision

For performing computer vision, several highly developed libraries are available, in this opportunity the libraries to be used will be *OpenCV* and *PCL* since they are suited to work along with ROS and allow to perform rapid prototypes of Computer Vision algorithms. The RGB and Depth Images provided by the Microsoft Kinect sensor and its open source drivers *Openni* will be processed to perform different task. In figure 17 can be seen an example of the provided images, see listing 8 for instructions.

```
netbook:
$ roslaunch rbx1_vision openni_node.launch
$ rosrun image_view image_view image:=/camera/rgb/image_color
$ rosrun image_view disparity_view image:=/camera/depth/disparity
```

Listing 8: Kinect drivers launch



(a) (b)

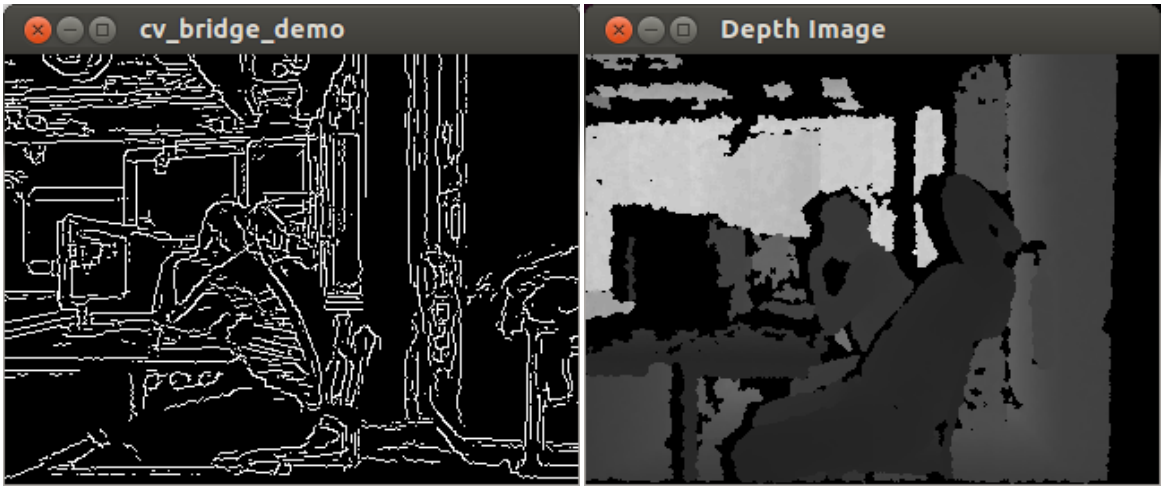
Figure 17: Kinect image types. (a) Color image, (b) Depth Image.

4.1 OpenCV

To process the ROS camera messages with *Open CV* the *cv_bridge package* is available. This package coverts the ROS camera images allowing to do simple processing and converting again to ROS Images to visualize the results. In figure 18 can be seen in action the some OpenCV filters for the RGB Image performing *Canny Edge Detection* and a simple visualization of the Depth camera, but converting between ROS and OpenCV formats.

```
netbook:
$ roslaunch rbx1_vision openni_node.launch
workstation:
$ rosrun rbx1_vision cv_bridge_demo.py
```

Listing 9: Open CV and ROS bridge



(a) (b)

Figure 18: Bridge demo. (a) Canny edge detection, (b) Depth Image.

4.1.1 Tracking

There exist many approaches to perform *Object Tracking* in Computer Vision, in the RBX1 package some of them have been implemented and the following ones were tested:

Face Detection

The face detector used in this tutorials uses the *Cascade Classifier with Haar-like features*, such classifier has already been trained exhaustively to be able to detect face under many different conditions, but it can be trained for many other objects and the trained data is stored in XML files. Results for the tracking can be seen in figure 19.

```
netbook:
$ roslaunch rbx1_vision openni_node.launch
worksataion:
$ roslaunch rbx1_vision face_detector.launch
```

Listing 10: Face detection

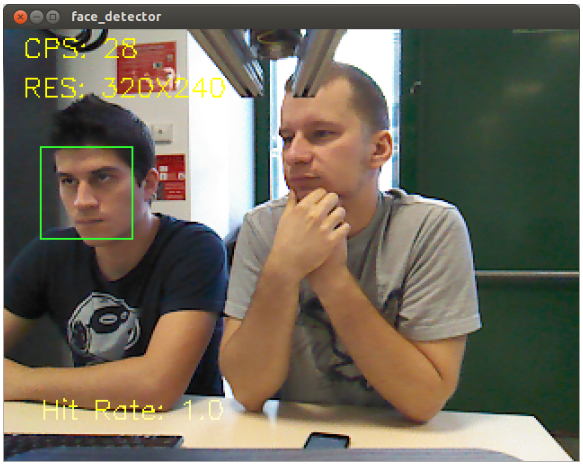


Figure 19: Face detection.

Features Tracking

In Computer Vision it is extremely important to extract representative points in the images that can be easily recognized one from the other, such points are known as key-points and from them are extracted features that are a sort of particular identification of every key-point. This approach uses *Good Features to Track* and looks for the key-points using *Harris Corner Detector*, therefore it extracts the features to be able to match them in an image sequence, in this case in the Kinect RGB camera video streaming. First thing to do is to define a *ROI (Region of Interest)* as can be seen the OpenCV allows to define it using a green rectangle.

```
netbook:
$ roslaunch rbx1_vision openni_node.launch
worksataion:
$ roslaunch rbx1_vision good_features.launch
```

Listing 11: Features tracking

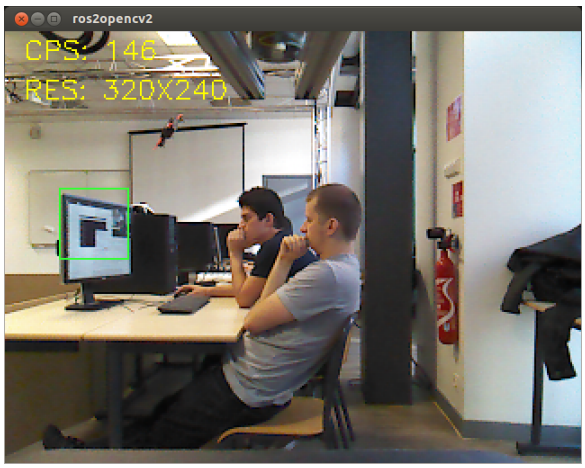
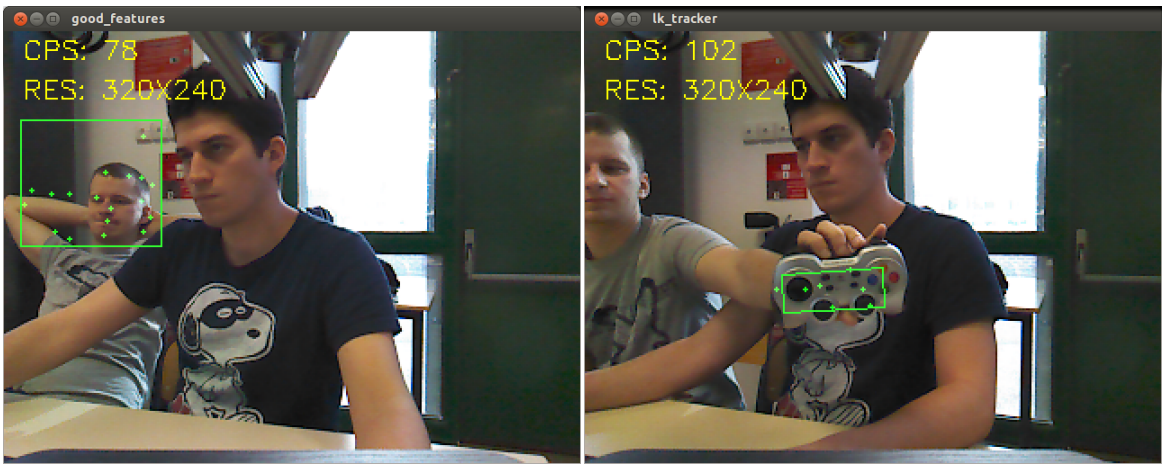


Figure 20: Setting Region of Interest.

After the ROI is defined key-points and their features are extracted and tracked in the video streaming. See figure 21(a) for Kirill's face tracking and (b) for joystick tracking.



(a) (b)

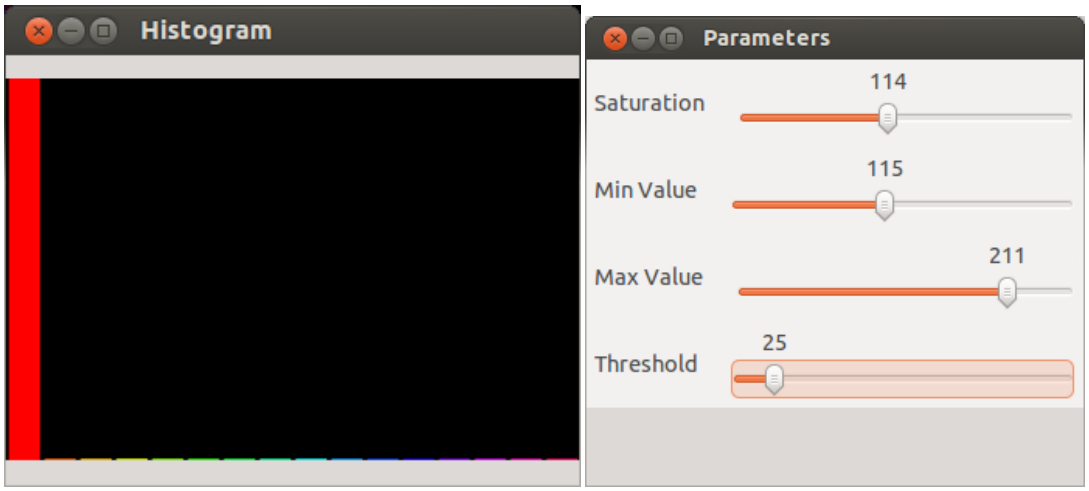
Figure 21: Features tracking using Good features to track. (a) Features of the face, (b) Features for the joystick.

Color Blob Tracking (CamShift)

The third and last approach tested consist in following objects by their color information, this methodology is recommended for objects with bright colors. This method is less computationally expensive than the previously presented, but has to be tuned for the particular object that wants to be tracked, it computes the histogram of the desired object and the levels of saturation, value and an arbitrary threshold. In figure 23 can be seen the histogram of the chosen ROI and the sliders for tuning the levels.

```
netbook:
$ roslaunch rbx1_vision openni_node.launch
worksataion:
$ roslaunch rbx1_vision camshift.launch
```

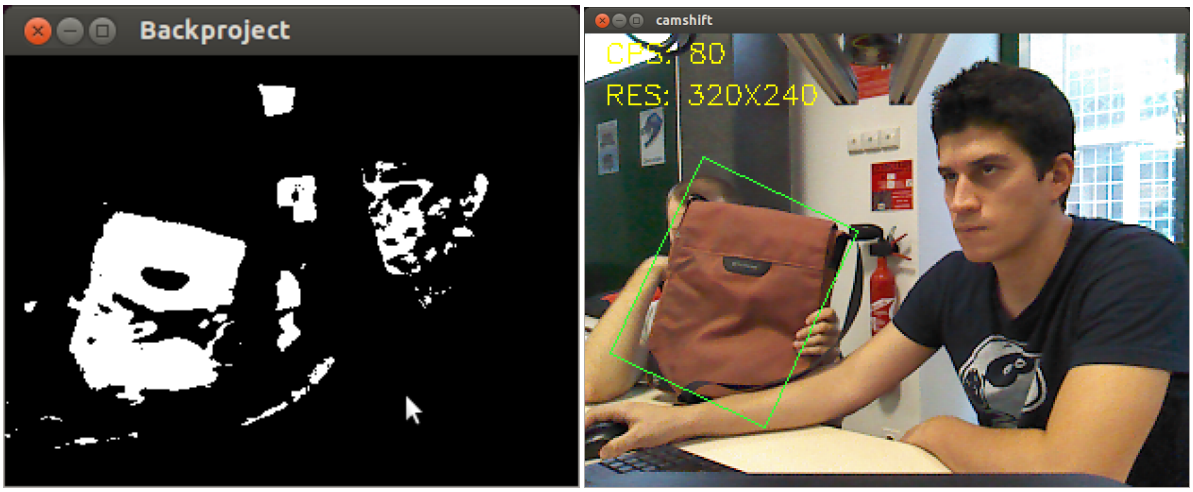
Listing 12: Color tracking



(a) (b)

Figure 22: Color Tracking parameters. (a) Histogram of the desired ROI, (b) Parameter values.

Once the object is segmented the algorithm is able to identify the object as the biggest object in the image.



(a) (b)

Figure 23: Color Tracking.

Skeleton

A pretty impresive application using the Kinect is the skeleton tracking where an algorithm identifies the shape of a human body and is able to track its different joints.

```
$ rosrun openni_tracker openni_tracker
$ rosrun rviz rviz -d `rospack find rbx1_vision`/skeleton_frames.rviz
$ roslaunch skeleton_markers markers_from_tf.launch
$ rosrun rviz rviz -d `rospack find skeleton_markers`/markers_from_tf.rviz
```

Listing 13: Color tracking

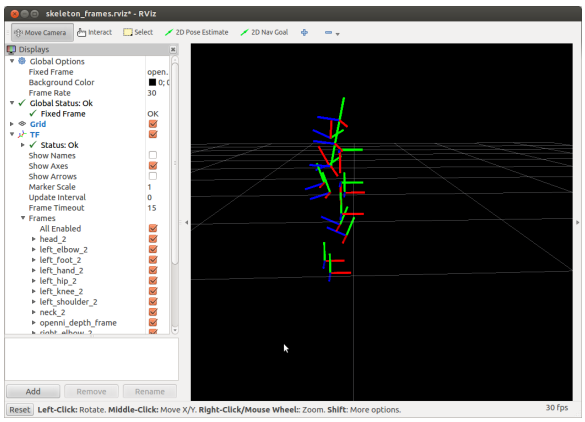
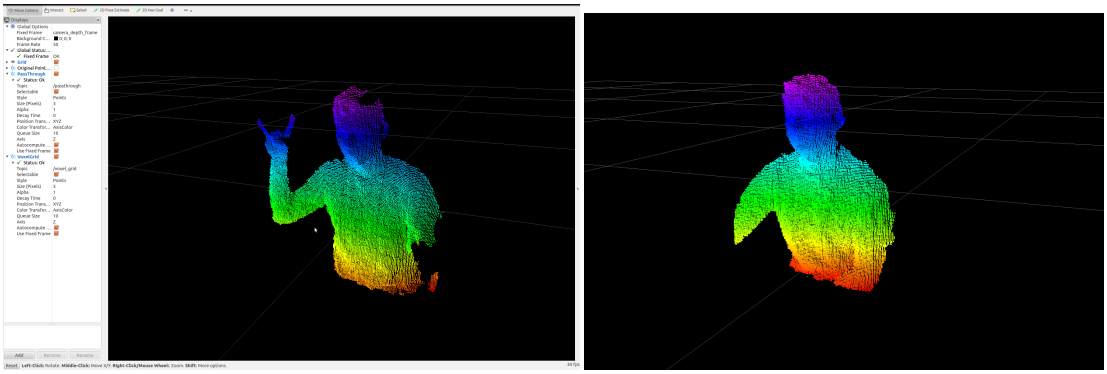


Figure 24: Skeleton tracking.

4.2 PCL

PCL stands for *Point Cloud Library* and similar to OpenCV is a set of tools to perform Computer Vision algorithms. It is a quite interesting tool when the Kinect Depth Images are represented in point clouds and some simple applications are presented below. See figure 25 for some point clouds displayed in RVIZ.



(a) (b)

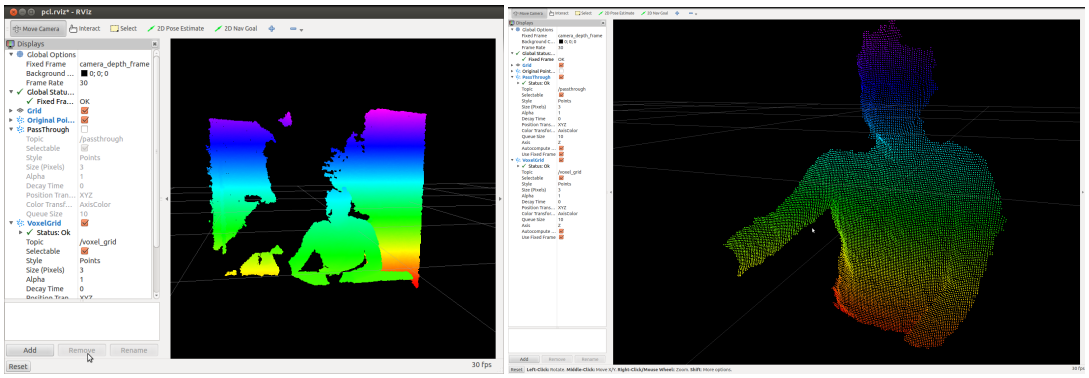
Figure 25: Point clouds in RVIZ. (a)Luis, (b) Kirill.

Passthrough Filter

As it can be seen in figure 26 (a) not all information contained in the point cloud is relevant for some applications, therefore a *Passthrough* can be applied for displaying the relevant information as in (b).

```
netbook:
$ roslaunch rbx1_vision openni_node.launch
workstation:
$ roslaunch rbx1_vision passthrough.launch
$ rosrun rviz rviz -d ‘rospack find rbx1_vision‘/pcl.rviz
```

Listing 14: Passthrough filter



(a) (b)

Figure 26: Passthrough Filtering.

Voxel Subsampling

Point clouds as the provided by the Kinect represent large amounts of information which can be redundant for certain applications or simply the data needs to be reduced due to computational requirements. To deal with these kind of issues the voxel subsampling is presented.

```
netbook:
$ roslaunch rbx1_vision openni_node.launch
workstation:
$ rosrun rviz rviz -d `rospack find rbx1_vision`/pcl.rviz
$ roslaunch rbx1_vision voxel.launch
```

Listing 15: Voxel subsampling

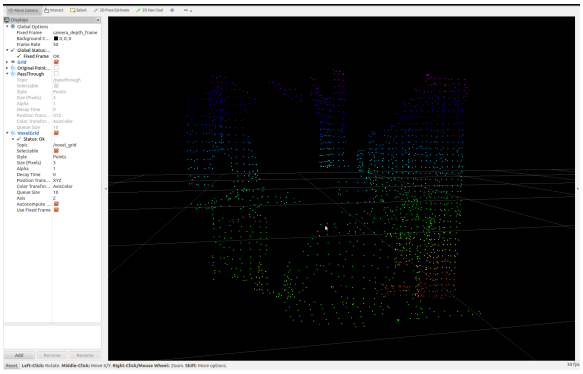
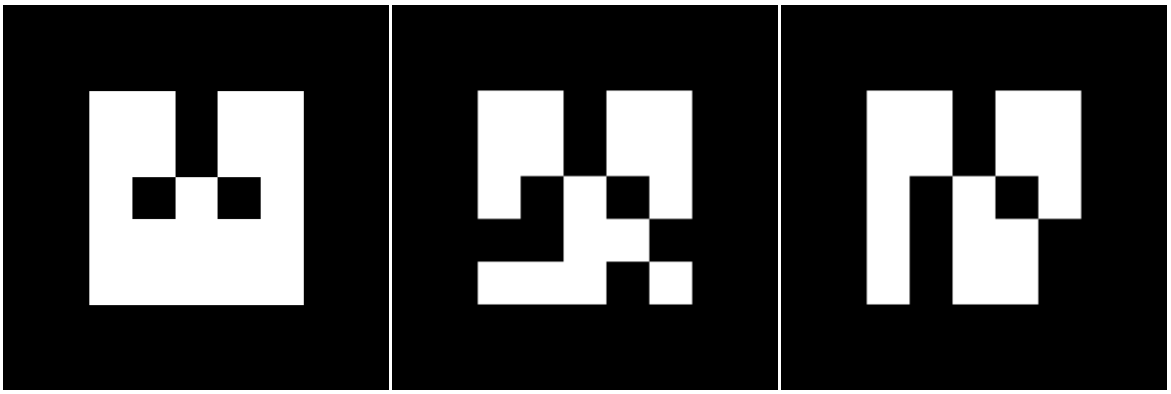


Figure 27: Voxel Subsampling.

4.3 AR Tags

Objects detection and recognition using cameras can be a difficult and ambiguous task due to the fact that the objects can be seen from infinite different points of view, this involves that the objects are seen projected, rotated and scaled and is even worse for mobile robots since they move in their environments. A robust way to give information to the robot using its camera is using *AR Tags*, these tags are sort of markers designed to be easily processed, and its detection is invariant to rotation, robust to different lighting condition and their shape allows to compute easily their projective transformation. In figure 28



(a) (b) (c)

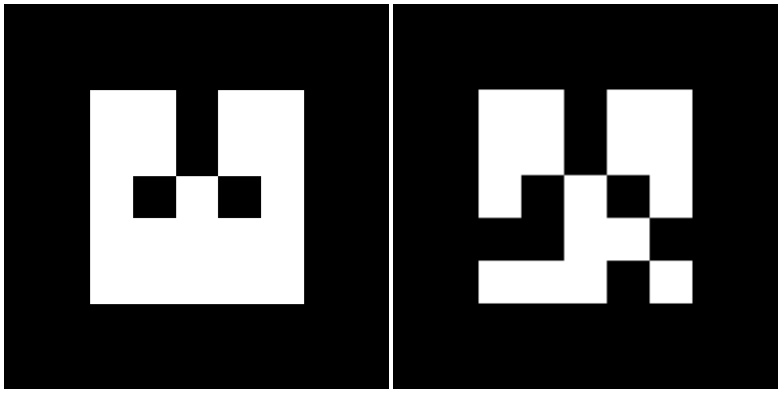
Figure 28: AR Tag.

This tags can be placed along the robot trajectory and recognized using the algorithms provided in the RBX2 package. First thing to do with the package is to create the tags running the the AR Tags creator as follows: *rosrun ar_track_alvar createMarker*, the tags can be associated to different kind of information and their size changed using the parameters, for instance if the tags want to be used for localization is better to use big tags and smaller tags in case the information wants to be seen in a specific place.

```
netbook:
$ roslaunch rbx2_vision openni_node.launch
$ roslaunch rbx2_ar_tags ar_indiv_kinect.launch
```

Listing 16: AR Tags launching

The program launched publishes whenever a tag or several tags are seen in the Kinect camera, using a topic array, for example: *rostopic echo /ar_pose_marker/markers[0]* indicates the first observed tag and its pose corresponding to its (X,Y,Z) position and the quaternions corresponding to the angle.



(a) (b)

Figure 29: AR Tags with real robot. (a) Maze, (b) Detection.

5 Task Execution

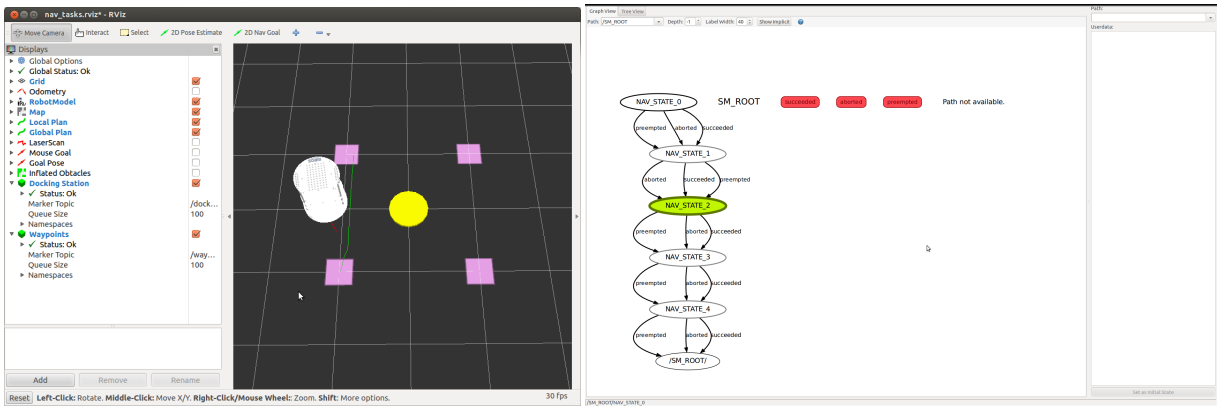
Till now the robot has executed different tasks individually or has made some small interaction between tasks, but still, in order to make real robotics applications, it is not practical to do it with a single script and making conditional statements to set up interactions between the different routines. To handle more complex behaviors, the Finite States Machine methodology is used, the *SMACH* library allows not only to set up the States Machine logic, but it is written in python and it can interact properly with ROS topics, services and so.

5.1 Patrolling with SMACH

The idea now is to give a set of destinations to the robot using finite state machine to emulate the previously seen *nav_square.py* scripts, even thought the result is the same the SMACH approach allows to integrate easily other routines. In figure 30 can be seen the fake robot with its four desired destinations and the corresponding diagram using the *SMACH Viewer*.

```
$ roslaunch rbx2_tasks fake_turtlebot.launch
$ rosrun rviz rviz -d `rospack find rbx2_tasks`/nav_tasks.rviz
$ rosrun smach_viewer smach_viewer.py
$ rosrun rbx2_tasks patrol_smach.py
```

Listing 17: Patroling using SMACH



(a) (b)

Figure 30: Patrolling a square using SMACH. (a) Fake robot patrolling, (b) SMACH Viewer.

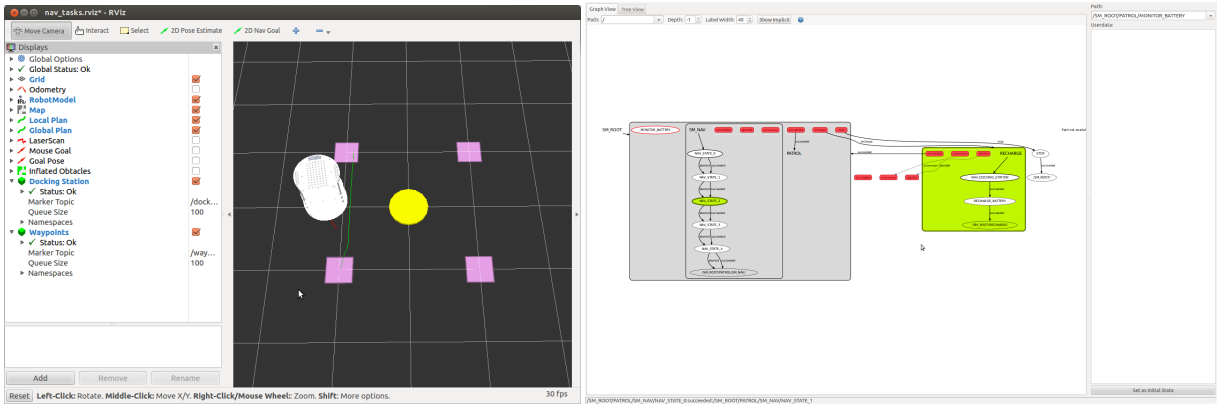
5.2 Patrolling and Battery Checking

States Machines are particularly useful when there has to be priority of one task over other, for instance in Mobile Robotics it is very important to constantly check the battery level while doing any other job so the robot does not run out of energy. The RBX2 package allows to perform navigation with the robot doing patrolling within the square path, but at the same time it constantly checks the level of a simulated battery, once the battery is below the desired level the robot goes straightly to the docking station postponing the patrolling task until the battery is recharged.

```
workstation:
$ roslaunch rbx1_bringup fake_turtlebot.launch
$ rosrun smach_viewer smach_viewer.py
$ rosrun rviz rviz -d `rospack find rbx2_tasks`/nav_tasks.rviz
$ rosrun rbx2_tasks patrol_smach_concurrence.py
```

Listing 18: Patroling and battery checking using SMACH

This topology shows that the main activity correspond to perform the navigation in the square which corresponds to *nav_sate_0 to nav_state_4*, at the same time the battery level is constantly being checked and has priority over the navigation task, this condition between navigation and battery monitoring is called concurrence. Once the battery level is below the threshold is the time to perform the recharge which consists in *navigate_docking_station* and the *recharge*.



(a) (b)

Figure 31: Patrolling and battery checking using SMACH. (a) Fake robot, (b) SMACH Concurrence scheme.

6 Final Project

For final project the idea is to integrate some of the different tools presented before to perform *Autonomous Navigation* and *Ar Tag Recognition*. Fourteen predefined points are set up in the maze, the robot will move randomly between these points while at the same time is constantly trying to detect the AR Tags with values between $61 - 65$, every time a tag is recognized a navigation point is placed 35 centimeters in front of it. When all tags are seen correctly, the robot will start navigating between the position placed in front of the tag. In `https://youtu.be/kZJ-9qH26AU` can be found a demonstration of the functioning of the project and the full project source code can be cloned or download from `https://github.com/hruslowkirill/turtlebot-patrol`.

For running the final project in the real robot two launch files have to be launched. The netbook launchfile concerns the robot drivers launching, the map of the maze and finally the Kinect drivers.

```
<launch>
  <include file="$(find turtlebot_le2i)/launch/remap_rplidar_minimal.launch"/>
  <include file="$(find rbx1_nav)/launch/tb_demo_amcl.launch">
    <arg name="map" value="final_map.yaml" />
  </include>
  <include file="$(find rbx2_vision)/launch/openni_node.launch"/>
</launch>
```

Listing 19: Launch file for the netbook

In the workstation the AR Tags recognizer is launched a long with the RVIZ configured to visualize the different markers.

```
<launch>
  <include file="$(find rbx2_ar_tags)/launch/ar_indiv_kinect.launch"/>
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find rbx1_nav)/nav.rviz"/>
</launch>
```

Listing 20: Launch file for workstation

Once the RVIZ is available, the pose estimation has to be done in order to help the *AMCL* to localize correctly and then the project itself can be run. In figure 32 can be seen the final project environment.

```
rosrun final_project_kl main.py
```

Listing 21: Running the main file of the project

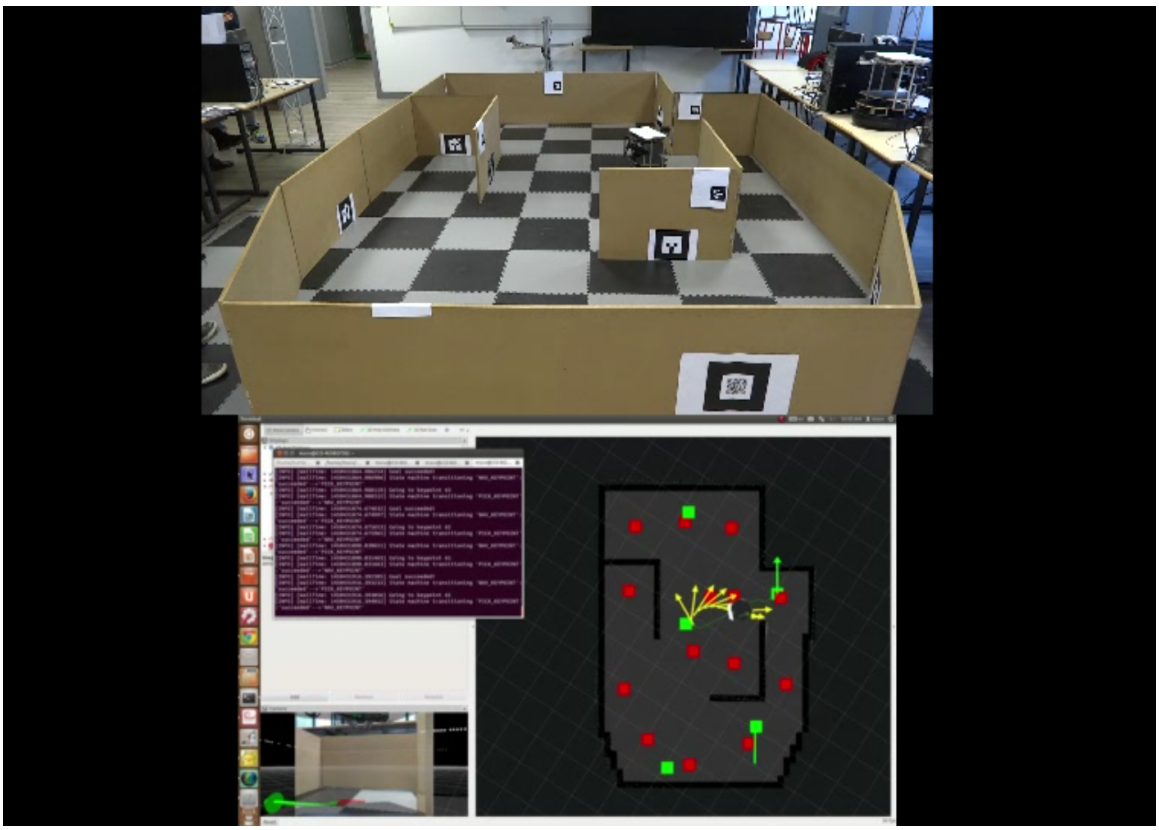


Figure 32: Final Project.

In figure 33 can be seen the Finite State Machine concerning the project, in the right side the concurrence between the random navigation and the AR tag detection can be appreciated, these task are done in parallel so the robot can find the tags all along the maze. In the left side the state machine to navigate along the found keypoints is seen.

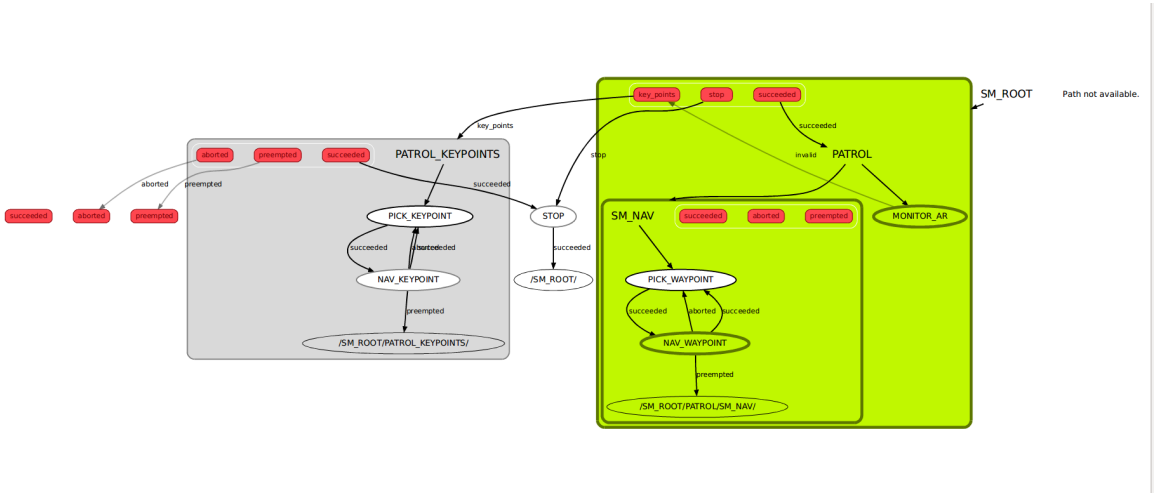


Figure 33: SMACH scheme for the Final Project.

6.1 Code Highlights

Since it is not practical just to show the whole code of the project, some important lines are going to be shown and briefly explained. Below can be found the concurrence defined for navigating and gathering the keypoint at the same time. The state *Monitor AR* constantly looks in the topic */ar_pose_marker*

```
# Create the nav_patrol state machine using a Concurrence container
self.nav_patrol = Concurrence(outcomes=['succeeded', 'key_points', 'stop'],
                default_outcome='succeeded',
                outcome_map = {'key_points' : {'MONITOR_AR':'invalid'}},
    child_termination_cb=self.concurrence_child_termination_cb,
                                        outcome_cb=self.concurrence_outcome_cb)
# Add the sm_nav machine and a AR Tag MonitorState to the nav_patrol machine
with self.nav_patrol:
  Concurrence.add('SM_NAV', Patrol().getSM())
  Concurrence.add('MONITOR_AR',MonitorState('/ar_pose_marker', AlvarMarkers, self.
    ar_cb))
```

Listing 22: Concurrence scheme

The random navigation and the navigation between the found keypoints is set up.

```
with self.sm_top:
  StateMachine.add('PATROL', self.nav_patrol, transitions={'succeeded':'PATROL', '
    key_points':'PATROL_KEYPOINTS', 'stop':'STOP'})
  StateMachine.add('PATROL_KEYPOINTS', PatrolThroughKeyPoints(self.keyPointManager
    ).getSM(), transitions={'succeeded':'STOP'})
  StateMachine.add('STOP', Stop(), transitions={'succeeded':''})
```

Listing 23: Random and keypoints navigation

Every time the */ar_pose_marker* topic publishes a detected marker, the function *ar_cb* is called, this function uses the class *keyPointManager* to check if the tag is between the desired range and if all the markers are detected and added to a list.

```
    def ar_cb(self, userdata, msg):
        if len(msg.markers)>0:
            rospy.loginfo('WE READ THE AR!: ' + str(msg.markers[0].id))
            rospy.loginfo(self.keyPointManager)
            for i in range(0,len(msg.markers)):
                if (self.keyPointManager.markerHasValidId(msg.markers[i])):
                    self.keyPointManager.add(msg.markers[i]);
                    if (self.keyPointManager.keyPointListComplete()):
                        return False
                    #self.sm_top.userdata.sm_ar_tag = msg.markers[i]
                    rospy.loginfo('NUMBER '+str(msg.markers[i].id)+' is detected!!
    ')
```

Listing 24: AR Tags detection

```
    def keyPointListComplete(self):
        if (len(self.keyPointList)==5):
            self.keyPointList.sort(key=lambda x: x.id, reverse=True)
            return True
        return False
    def markerHasValidId(self, marker):
        rospy.loginfo("marker.id: "+str(marker.id))
        if (marker.id>=61) and (marker.id<=65):
            return True
        return False
    def add(self, marker):
        #rospy.loginfo(marker)
        for i in range(len(self.keyPointList)):
            if (self.keyPointList[i].id==marker.id):
                return
        position = self.transformMarkerToWorld(marker)
        k = KeyPoint(marker.id, Point(position[0], position[1], position[2]),
    Quaternion(0., 0., 0., 1.))
        self.keyPointList.append(k)
        self.addWaypointMarker(k)
        rospy.loginfo('position')
        rospy.loginfo(k.pose)
        pass
```

Listing 25: Keypoint manager important functions

Every time the marker is valid, a keypoint is defined 35 centimeters ahead of the tag, this is done using the found frame of the tag multiplying the following transformation matrices.

$$TAG = \begin{pmatrix} cos\theta & -sin\theta & 0 & t_x \\ sin\theta & cos\theta & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0.35 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
    def shiftPoint(self, position, quaternion):
        rospy.loginfo('position' + str(position))
        try:
            euler = euler_from_quaternion(quaternion)
            rospy.loginfo('euler: ' + str(euler))
            yaw = euler[2]
            tf_mat = np.array([[np.cos(yaw), -np.sin(yaw), 0, position[0]],[np.sin
    (yaw), np.cos(yaw), 0, position[1]],[0, 0, 1, 0],[0, 0, 0, 1]])
            displacement = np.array([[1, 0, 0, 0],[0, 1, 0, 0.35],[0, 0, 1, 0],[0,
     0, 0, 1]])
            point_map = np.dot(tf_mat, displacement)
            position = (point_map[0,3], point_map[1,3], 0)
            rospy.loginfo('final position' + str(position))
        except Exception as inst:
            print type(inst)     # the exception instance
            print inst.args      # arguments stored in .args
            print inst
        return position
```

Listing 26: Keypoint placement

The two state machines concerning the random movement and the movement through the found keypoints are shown below, they use the methods *PickWaypoint* and *PickWaypointKeyPoint* from the random list and from the list created by the perceived tags. Finally they navigate to the chosen point using the classes *Nav2Waypoint* which uses *move_base* as explained before.

```

class Patrol(State):
    def __init__(self):
        State.__init__(self, outcomes=['succeeded','aborted','preempted'])
        setup_task_environment(self);
        # Initialize the navigation state machine
        self.sm_nav = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])
        self.sm_nav.userdata.waypoints = self.waypoints
        with self.sm_nav:
            StateMachine.add('PICK_WAYPOINT', PickWaypoint(),
                             transitions={'succeeded':'NAV_WAYPOINT'},
                             remapping={'waypoint_out':'patrol_waypoint'})

            StateMachine.add('NAV_WAYPOINT', Nav2Waypoint(),
                             transitions={'succeeded':'PICK_WAYPOINT',
                                          'aborted':'PICK_WAYPOINT',
                                          'preempted':''},
                             remapping={'waypoint_in':'patrol_waypoint'})

        pass
    def move_base_result_cb(self, userdata, status, result):
        pass
    def execute(self, userdata):

        rospy.loginfo("Patroling")
        while 1>0:
            if self.preempt_requested():
                self.service_preempt()
                return 'preempted'
            #rospy.SpinOnce();
        return 'succeeded'
```

```
32      def getSM(self):
33          return self.sm_nav
34
35  class PatrolThroughKeyPoints(State):
36      def __init__(self, keyPointManager):
37          State.__init__(self, outcomes=['succeeded','aborted','preempted'])
38          self.keyPointManager = keyPointManager
39          #setup_task_environment(self);
40          # Initialize the navigation state machine
41          self.sm_nav = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])
42          rospy.loginfo("PatrolingTHoughKeypoints!!!")
43          #self.sm_nav.userdata.waypoints = self.keyPointManager.getWaypoints()
44          with self.sm_nav:
45              StateMachine.add('PICK_KEYPOINT', PickWaypointKeyPoint(self.
    keyPointManager),
46                               transitions={'succeeded':'NAV_KEYPOINT'},
47                               remapping={'waypoint_out':'patrol_waypoint'})
48
49              StateMachine.add('NAV_KEYPOINT', Nav2Waypoint(),
50                               transitions={'succeeded':'PICK_KEYPOINT',
51                                            'aborted':'PICK_KEYPOINT',
52                                            'preempted':''},
53                               remapping={'waypoint_in':'patrol_waypoint'})
54
55          pass
56      def move_base_result_cb(self, userdata, status, result):
57          pass
58      def execute(self, userdata):
59          self.sm_nav.userdata.waypoints = self.keyPointManager.getWaypoints()
60          rospy.loginfo("PatrolingTHoughKeypoints")
61          rospy.loginfo(self.sm_nav.userdata.waypoints)
62
63          rospy.loginfo(self.keyPointManager)
64          while 1>0:
65              if self.preempt_requested():
66                  self.service_preempt()
67                  return 'preempted'
68              #rospy.SpinOnce();
69          return 'succeeded'
70      def getSM(self):
71          return self.sm_nav
```

Listing 27: Patrolling State Machines

Below can be found the classes *PickWaypoint*, *PickWaypointKeyPoint* and *Nav2WayPoint*.

```
1
2   class PickWaypoint(State):
3       def __init__(self):
4           State.__init__(self, outcomes=['succeeded'], input_keys=['waypoints'],
    output_keys=['waypoint_out'])
5
6       def execute(self, userdata):
7           waypoint_out = randrange(len(userdata.waypoints))
8
9           userdata.waypoint_out = userdata.waypoints[waypoint_out]
10
11          rospy.loginfo("Going to waypoint " + str(waypoint_out))
12
13          return 'succeeded'
14
15  class PickWaypointKeyPoint(State):
16      def __init__(self, keyPointManager):
17          State.__init__(self, outcomes=['succeeded'], input_keys=['waypoints'],
    output_keys=['waypoint_out'])
18          self.keyPointManager = keyPointManager
19          self.currentKeyPoint = 0
```

```
    def execute(self, userdata):
        if (self.currentKeyPoint==len(self.keyPointManager.keyPointList)):
            self.currentKeyPoint = 0
            #return 'finished'

        userdata.waypoint_out = self.keyPointManager.keyPointList[self.
    currentKeyPoint].pose

        rospy.loginfo("Going to keypoint " + str(self.keyPointManager.keyPointList
    [self.currentKeyPoint].id))
        self.currentKeyPoint = self.currentKeyPoint+1

        return 'succeeded'

class Nav2Waypoint(State):
    def __init__(self):
        State.__init__(self, outcomes=['succeeded','aborted','preempted'],
                       input_keys=['waypoint_in'])

        # Subscribe to the move_base action server
        self.move_base = actionlib.SimpleActionClient("move_base", MoveBaseAction)

        # Wait up to 60 seconds for the action server to become available
        self.move_base.wait_for_server(rospy.Duration(60))

        rospy.loginfo("Connected to move_base action server")

        self.goal = MoveBaseGoal()
        self.goal.target_pose.header.frame_id = 'map'

    def execute(self, userdata):
        self.goal.target_pose.pose = userdata.waypoint_in

        # Send the goal pose to the MoveBaseAction server
        self.move_base.send_goal(self.goal)

        if self.preempt_requested():
            self.move_base.cancel_goal()
            self.service_preempt()
            return 'preempted'

        # Allow 1 minute to get there
        finished_within_time = self.move_base.wait_for_result(rospy.Duration(60))

        # If we don't get there in time, abort the goal
        if not finished_within_time:
            self.move_base.cancel_goal()
            rospy.loginfo("Timed out achieving goal")
            return 'aborted'
        else:
            # We made it!
            state = self.move_base.get_state()
            if state == GoalStatus.SUCCEEDED:
                rospy.loginfo("Goal succeeded!")
            return 'succeeded'
```

Listing 28: Chosing points and navigating

7 Conclusions

- ROS is a pretty versatile platform to develop robotics projects thanks to it set tools and packages, its architecture allows to access easily to the different devices and makes a robust integration of them.

- Computer Vision and Mobile Robotics are still a challenging topic and these kind of open source technologies make it feasible to perform rapid prototyping and implementation of state of the art algorithms in less time.
- Finite State Machines are a perfect tool to develop complex behaviors of robots.
- Augmented Reality Tags (AR Tags) detection is pretty robust to different conditions and therefore are suitable to perform task in real environments when is needed to give information to the robot using the camera.

8 Bibliography

References

Goebel, P. (2013). *ROS By Example I.* Lulu.

Goebel, P. (2014). *ROS By Example II.* Lulu.

Seulim, R. (2015). Robotic's project. University Lectures.

Various (2015). Ros tutorials @ONLINE.