Politecnico di Torino

Master of Science in Mechatronic Engineering

Development of new functionalities for the humanoid robot ROMEO

Master Thesis

Benoit HEINTZ – 217115

benoit.heintz@ensil.unilim.fr

July 2017

POLITECNICO DI TORINO Supervisor : Prof. Basilio BONA

Acknowledgements

I want to thanks all the people that participated in many different ways to the success of my internship and more particularly :

The research laboratory INRIA which gave me the opportunity to realize my internship in good conditions, notably **M. François CHAUMETTE**, team leader of the project team LAGADIC.

My internship supervisor **M. Fabien SPINDLER,** for the trust he put in me in letting me manage this project and for all the numerous and valuable advice he gave me.

My academic advisor from the ENSIL, **M. Stéphane RENAULT** and my academic advisor from the POLITECNICO DI TORINO **M. Basilio BONA** for the supervision that they did during this internship.

Giovanni CLAUDIO for supporting and helping me during the entirety of the duration of this internship.

All the members of the LAGADIC team, Bryan, Noël, Don Jovan, Aly, Suman, Nicolas, Pierre, Lesley, Quentin, Jason, Fabrizio, Thomas, Souriya et Aurélien.

And lastly, Hélène de la RUEE (Assistant) for all her help during the internship.

Summary

Illustrations' index

Introduction

Robotics is a field that is evolving and developing very quickly since the last years, notably in the field of humanoid robotic. The ALDEBARAN company, which has been brought since 2012 by the Japanese company SOFTBANK ROBOTICS, realized three humanoid robots. Firstly, with NAO, they allowed a lot of students and teachers to discover and to get familiar with robots, then with PEPPER, they created an emotional robot able to detect and react to the emotions of the person facing it. And lastly, with ROMEO, they want to create a humanoid robot able to help us, humans, in our daily life. But ROMEO is still only a prototype, and not yet commercialized. There exist only five prototypes in the whole world, and one can be found in the INRIA laboratory, at Rennes (France). The LAGADIC team is currently working with it, in the field of vision.

As part of my studies in engineering school, specialized in mechatronics, I wanted to realize the master thesis at the end of my master in a laboratory that is preoccupied by the questions of tomorrow in the field of humanoid robotic and vision. Indeed, my goal is to work in this domain of robotics that I particularly appreciate. That is why I joined the LAGADIC team from the INRIA laboratory, I wanted to learn from them and to develop a project in its entirety.

The possibilities for adding functionalities to the humanoid robot ROMEO being great, I had for mission, at the beginning, to realize a simulation system that would test the new functions for the robot that are developed by the different teams without having to realize the tests on the real robot. Following some problems that prevented us from continuing on this, I then had to carry out a program that would allow ROMEO to open and close a door.

In a first step we will see a few elements about the LAGADIC team and we will make a quick overview of ROMEO and the different softwares that I used during my internship. Then we will discover the different missions that were given to me, with an important part on the simulation and why we had to stop without finishing this project. Then we will make a further analysis on the second project that I had to realize, with the help of Giovanni and Fabien. For this analysis, we will divide my presentation in two parts, one being the localization of a door handle and the other one being the different algorithms that I used to make ROMEO moves and grabs the said door handle.

I. Work environment

The LAGADIC research project bilocalised in Rennes (France) and Sophia-Antipolis (France) exploits a number of specific materials on which are validated all the research work developed in closed-loop visual servoing, monitoring and active perception. From a software point of view, these materials are interfaced in ViSP, a library developed by the LAGADIC team and distributed as free software, and also in ROS.

In September 2014, the LAGADIC team was the first of four European laboratories to be equipped with ROMEO. Today, this platform is used to validate their research. As soon as it arrived, from their work in visual control and visual monitoring, they developed a first experiment to capture an object (a square box) in order to give it to a human.

1. ROMEO

ROMEO (Illustration 1) is a 147 cm tall humanoid robot, designed to explore and further research into assisting elderly people and those who are losing their autonomy. ROMEO is the fruit of collaboration between numerous French and European laboratories and institutions. His size was determined so as to enable him to open doors, climb stairs and reach objects on a table.

ROMEO is equipped with four RGB cameras, one on each eye, which can move separately according to where it wants to look, and two others, that are fixed on its forehead, just on top of its eyes. These last two cameras are fixed in order to be able to use them as a stereo system, to obtain a depth map. But for now, this system is not yet implemented, and we used a 3D camera, as it is explained in part 3.1.

ROMEO is composed of 37 joints, 4 microphones, 2 loudspeakers, some tactile sensor on its head, an inertial central, and some force sensitive resistors under each foot (Illustration 2). With the microphones and the loudspeakers, ROMEO is able to discuss and interact fluently with any human that is speaking with it. An interesting part of ROMEO is that the motors used for the legs, and for the hands are made with cables, in order to give Romeo a more human-like behavior, as the cables are supposed to be the equivalent of our ligaments.

ILLUSTRATION 1: ROMEO

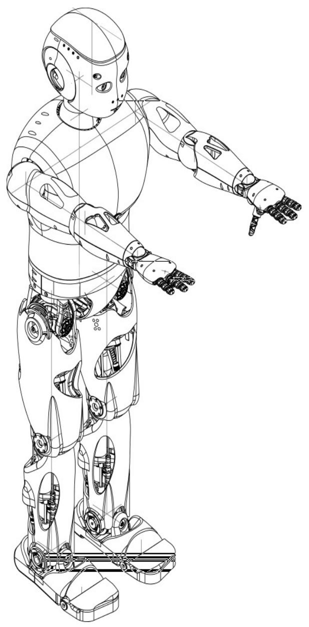


ILLUSTRATION 2: SCHEME OF ROMEO'S HARDWARE

1

2. Vision Software: ViSP & PCL

2.1. ViSP

ViSP, which stands for Visual Servoing Platform, is a modular cross platform library that allows prototyping and developing applications using visual tracking and visual servoing technics at the heart of the researches done by Inria Lagadic team. ViSP is able to compute control laws that can be applied to robotic systems. It provides a set of visual features that can be tracked using real time image processing or computer vision algorithms (Illustration 3). ViSP provides also simulation capabilities and can be useful in robotics, computer vision, augmented reality and computer animation.

ViSP provides simple ways to integrate and validate new algorithms with already existing tools. It follows a module-based software engineering design where data types, algorithms, sensors, viewers and user interaction are made available.

As it is written in C++, ViSP is based on open-source cross-platform libraries (such as OpenCV) and builds with CMake. Several platforms are supported, including OSX, Windows and Linux.

ViSP is used in this project only to realize a blob tracking, a filter on 2D images with a real-time constraint, and various dilatations and erosions operations.

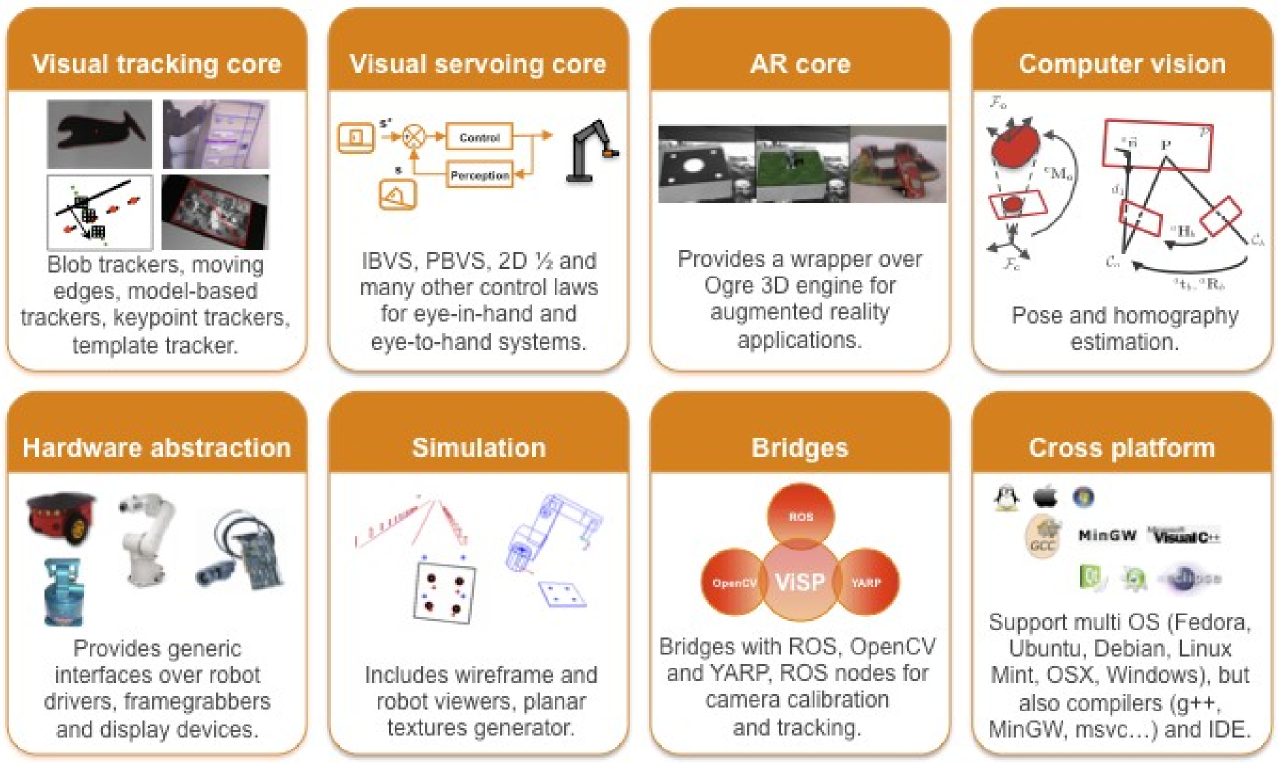

Visual tracking core
Blob trackers, moving edges, model-based trackers, keypoint trackers, template tracker.
Visual servoing core
Control
Perception
IBVS, PBVS, 2D ½ and many other control laws for eye-in-hand and eye-to-hand systems.
AR core
Provides a wrapper over Ogre 3D engine for augmented reality applications.
Computer vision
Pose and homography estimation.
Hardware abstraction
Provides generic interfaces over robot drivers, framegrabbers and display devices.
Simulation
Includes wireframe and robot viewers, planar textures generator.
Bridges
ROS
OpenCV
ViSP
YARP
Bridges with ROS, OpenCV and YARP, ROS nodes for camera calibration and tracking.
Cross platform
GCC
MinGW
Visual C++
Support multi OS (Fedora, Ubuntu, Debian, Linux Mint, OSX, Windows), but also compilers (g++, MinGW, msvc...) and IDE.

ILLUSTRATION 3: VISP FEATURES

2.2. PCL : Point Cloud Library

A point cloud is a data structure used to represent points of several dimensions, which are generally three-dimensional and are more often the geometric coordinates X, Y, and Z. But these points can also have a fourth dimension if for example, a color information is added to these points.

The Point Cloud Library (PCL) is a large free project allowing to use and modify 2D and 3D images as well as point clouds. It contains many state-of-the-art algorithms, including filters, surface recognition, segmentation, etc.

From this library, the following algorithms have been used : a point reduction filter, a filter to select a region of interest, and a plane segmentation.

Here is a brief explanation of the different algorithms :

For the filter to select a region of interest, the principle is simple: from some limits fixed in different coordinates, the algorithm eliminates all points not belonging to these boundaries.

For example, with 0.02m and 0.08m at the minimum and maximum boundaries along the Z coordinate, the following cloud of points (Illustration 4) becomes the following (Illustration 5). (See code in annex 1).

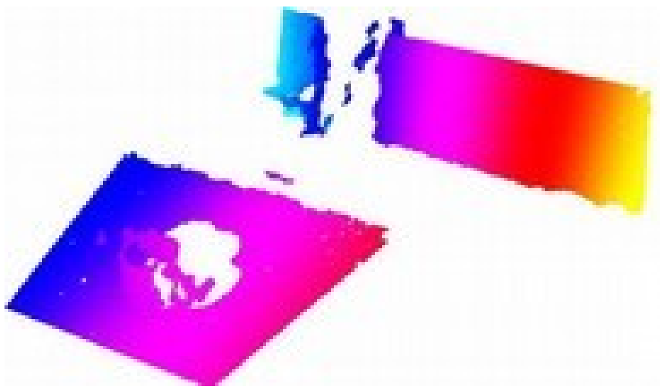


ILLUSTRATION 4: POINTCLOUD TOTAL

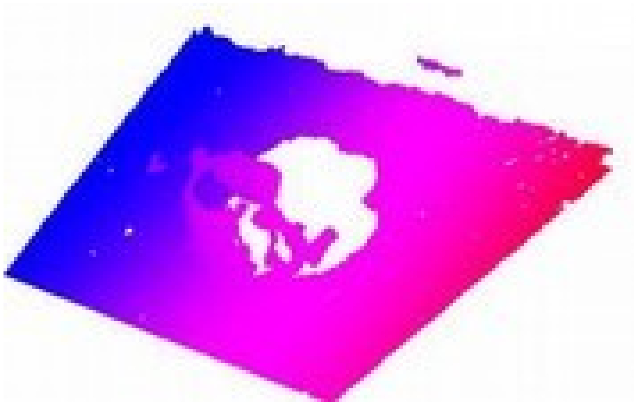


ILLUSTRATION 5: POINTCLOUD FILTERED

For the reduction filter, a Voxel grid filter is used. It creates a *3D voxel grid* (a voxel grid is a sort of a set of small 3D boxes in space) over the input point cloud data. Then, in each *voxel* (i.e., 3D box), all the points present will be approximated (i.e., *downsampled*) with their centroid. This approach is a bit slower than approximating them with the center of the voxel, but it represents the underlying surface more accurately. (Illustration 6). (See code in annex 2).

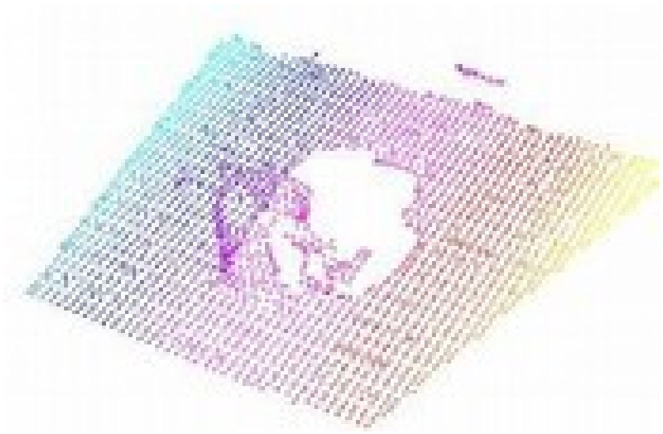


ILLUSTRATION 6: POINTCLOUD AFTER A VOXEL GRID FILTER

Finally, a plane segmentation algorithm is used. It is based on a RANSAC, RANdom SAmple Consensus, and determines if each point is part of the inliers or the outliers of the model of a plane. This makes it possible to obtain the equation and the set of inliers of the detected plane from the whole point cloud. The inliers are points considered to have geometric coordinates sufficiently close to those of the model of a plane to be considered as part of said plane, while the outliers are points outside the detection threshold. (Illustration 7) (See code in annex 3).

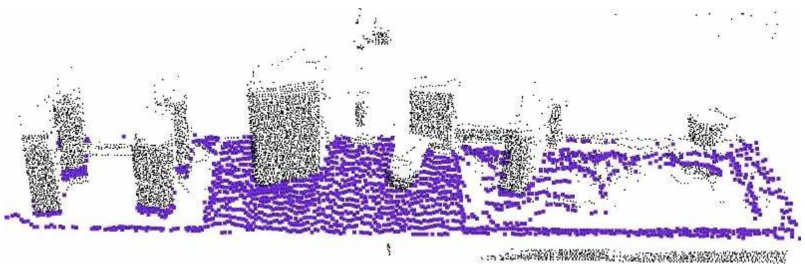


ILLUSTRATION 7: PLANE SEGMENTATION: INLIERS IN VIOLET AND OUTLIERS IN GREY

3. Softbank Robotics Software : NAOqi & Choregraphe

3.1. NAOqi Framework

NAOqi is the main software that runs on the robot of Softbank Robotics and controls it. The NAOqi Framework is the programming framework used to program NAO, ROMEO and PEPPER. It answers to common robotics needs including: parallelism, resources, synchronization, events. This framework allows homogeneous communication between different modules (motion, audio, video), homogeneous programming and homogeneous information sharing.

The framework :

- ✔ is cross-platform, which means that it is possible to develop with it on Windows, Linux or Mac.
- ✔ is cross-language, with an identical API for both C++ and Python.
- ✔ also provides introspection, which means the framework knows which functions are available in the different modules and where.

It is cross-language as the programming methods are exactly the same, for C++ and Python. All the existing API can be indifferently called from all the supported languages.

Introspection is the foundation of robot API, capabilities, monitoring and action on monitored functions. The robot knows all the available API functions. Unloading a library will automatically remove the corresponding API functions. A function defined in a module can be added in the API with a BIND_METHOD (defined in ALModule.h).

When binding a function - just three source code lines - you automatically benefit from the following features :

- ✔ Call function in both C++ and Python
- ✔ Know function if the function is being executed
- ✔ Execute function locally or remotely (from a computer or another robot)
- ✔ Call wait, stop, isRunning on functions

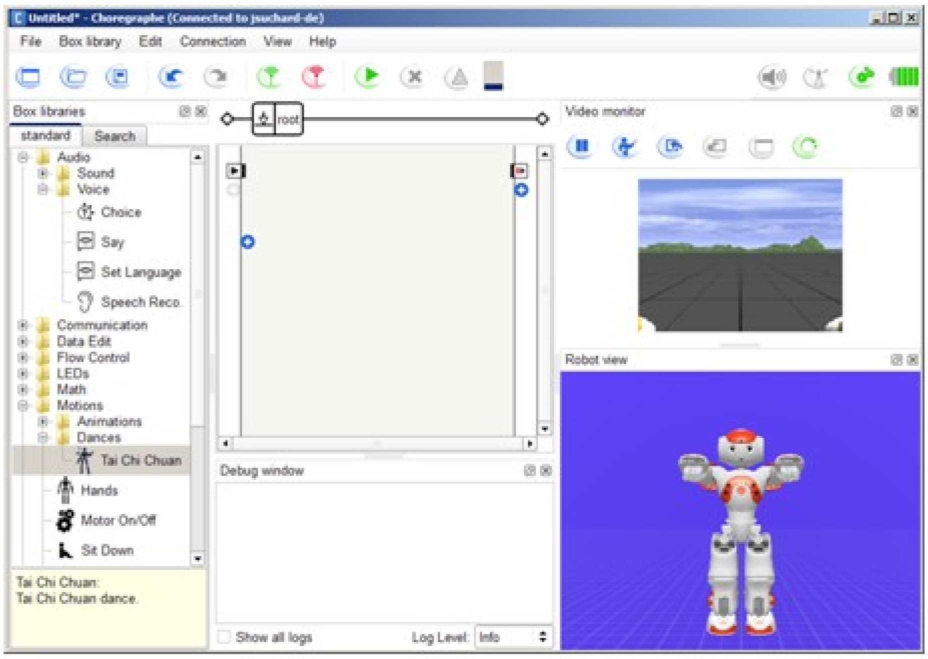
3.2. Choregraphe

Choregraphe is a multi-platform desktop application, developed by Softbank Robotics Europe (Illustration 8) for the robots that are manufactured by the same company, which allows to :

- ✔ Create animations, behaviors and dialogs for NAO, PEPPER or ROMEO,
- ✔ Test them on a simulated robot, or directly on a real one,
- ✔ Monitor and control your robot,
- ✔ Creating and packaging complete applications,
- ✔ Enrich Choregraphe behaviors with your own Python code,
- ✔ Designing sophisticated verbal interactions, thanks to QiChat, the human-robot dialogue design language,

This software allows people to create applications containing dialogs, services and powerful behaviors, such as interaction with people, dance, e-mails sending, without writing a single line of code.

During this internship this software was mainly used for monitoring and controlling ROMEO, including during the first part of my work at the lab, where I had to simulate ROMEO (See section 2.1).



Tai Chi Chuan:
Tai Chi Chuan dance.

ILLUSTRATION 8: CHOREGRAPHE CONNECTED TO A NAO ROBOT

4. Other Middleware & Software : ROS & V-REP

4.1. ROS

ROS is an operating system for robotics that was created a decade ago. It's an Open source system, easy to use thanks to the tutorials available on the site, in C ++ and Python. It allows to develop software for robotics, and is now supported on more than 75 robots, including the NAO robot.

Each application is composed of independent programs, the nodes, which communicate between them by the exchange of messages, topics, and under the supervision of a master program, the roscore.

More precisely, ROS presents a flexible inter-process and inter-machine communication architecture. ROS processes are called nodes. A node may correspond, for example, to a sensor, a motor, a processing and monitoring algorithm, etc. Each node that starts is declared to the master.

Each node can communicate with other nodes. The exchange of information takes place asynchronously via topics. They are systems of transport of information based on the system of subscription / publication. A topic is somehow an asynchronous information bus much like an RSS feed. This notion of many-to-many asynchronous buses is essential in the case of a distributed system. A topic is typed: the type of information that is published (the message) is always structured in the same way. Nodes send or receive messages on topics.

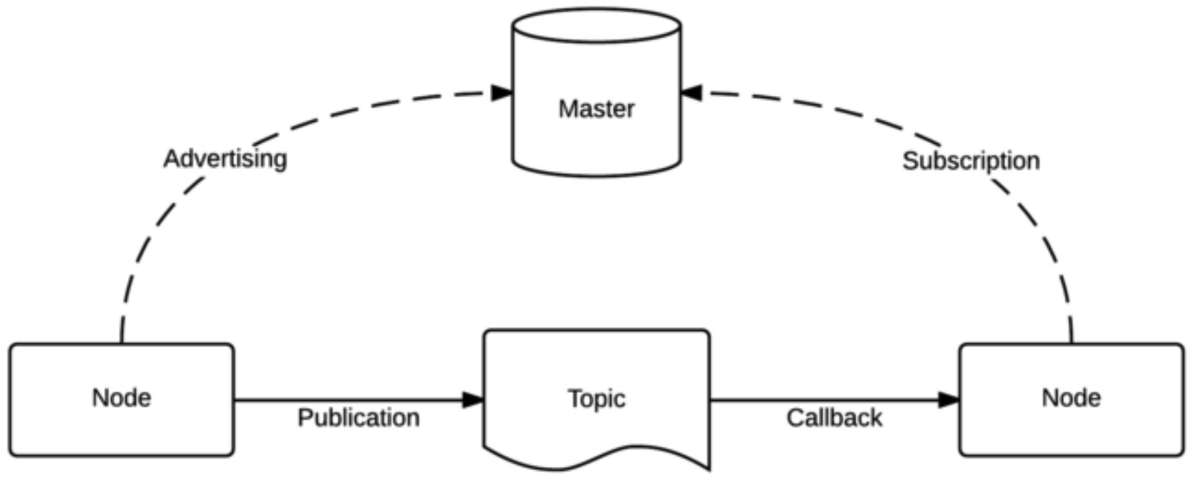
The connection between the nodes is managed by a master. The master is a service of declaration and registration of the nodes which allows nodes to know each other and exchange information.

The connection between nodes follows the following process : a first node notifies the master program that it has data to share. A second node notifies the master program that it wishes to have access to a given data. A connection between these two nodes is then created. The first node can send data to the second. So one or more nodes can publish informations on a topic and one or more nodes will be able to read the information on this topic (Illustration 9).

A node that publishes data is called a publisher. A node that subscribes to data is called a subscriber. A node can be both at once.

Since the messages sent on the topics are mostly standardized, the system is flexible. ROS allows inter-machine communication, and even nodes that are running on separate machines, but that have the knowledge of the same master, can communicate seamlessly to the user.

ILLUSTRATION 9: ROS ARCHITECTURE



Programming in ROS is done either in C ++ or in Python, but I only used C++ for my work. This was also the first time that I had to use this programming language, so it allowed me to gain a significant experience with C++.

ROS is used in this project to make the communication in the simulation, and to recover the point cloud emitted by the 3D camera Intel SR300. Illustration 10 presents an example of a set of points recovered with the camera, and transcribed on Rviz. Recovery of a set of points is done using a self-made program, which was completed and improved throughout the project.

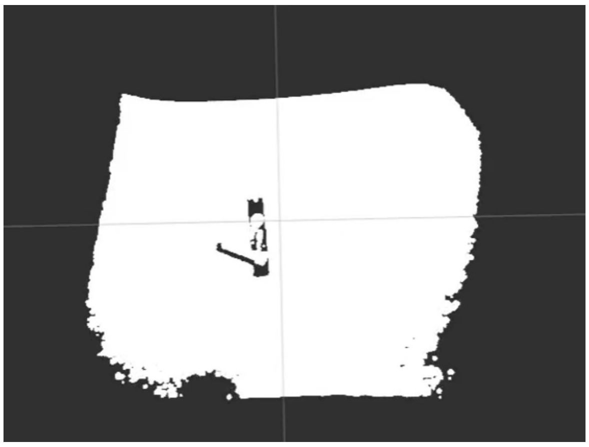


ILLUSTRATION 10: POINTCLOUD SEND BY A 3D CAMERA

Rviz is a 3D visualization tool that allows the display of data and informations from ROS. Here, (Illustration 11) Rviz displays a model of the NAO robot that is controlled via a bridge with ROS.

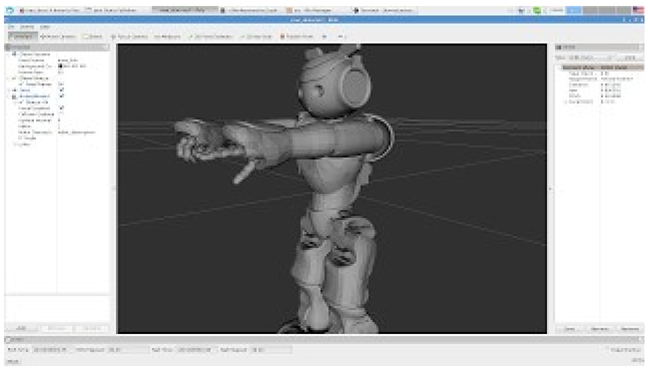


ILLUSTRATION 11: RVIZ SOFTWARE

4.2. V-REP

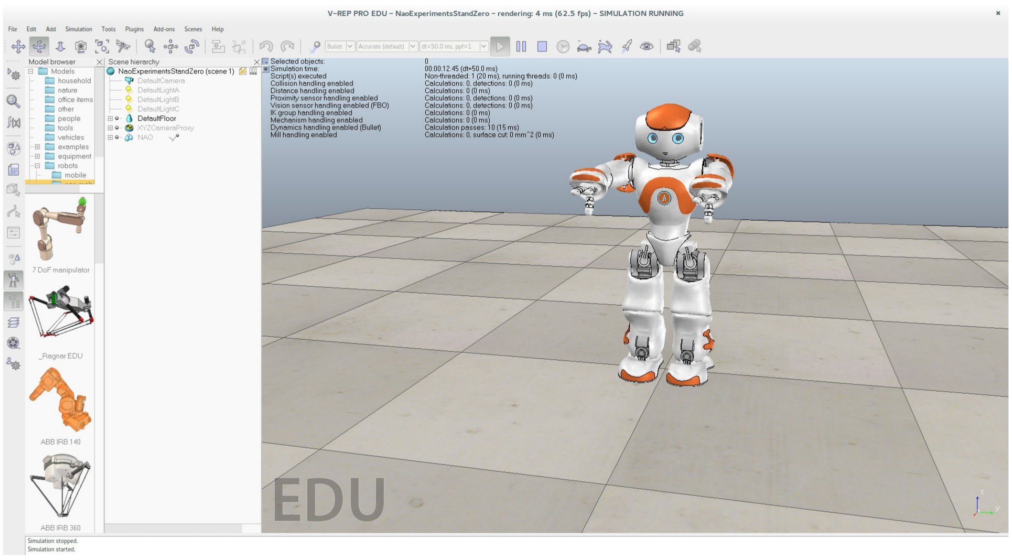
The robot simulator V-REP, (Illustration 12) with integrated development environment, is based on a distributed control architecture: each object/model can be individually controlled via an embedded script, a plugin, a ROS node, a remote API client, or a custom solution. This makes V-REP very versatile and ideal for multi-robot applications. Controllers can be written in C/C++, Python, Java, Lua, Matlab, Octave or Urbi.

Following are just a few of V-REP's applications:

- ✔ Simulation of factory automation systems
- ✔ Remote monitoring
- ✔ Hardware control
- ✔ Fast prototyping and verification
- ✔ Safety monitoring
- ✔ Fast algorithm development
- ✔ Robotics related education
- ✔ Product presentation

V-REP can be used as a stand-alone application or can easily be embedded into a main client application: its small footprint and elaborate API makes V-REP an ideal candidate to embed into higher-level applications. An integrated Lua script interpreter makes V-REP an extremely versatile application, leaving the freedom to the user to combine the low/high-level functionalities to obtain new high-level functionalities.

During my internship, after a brief overview of the other simulators that exist, (Gazebo, MORSE, webots, etc.) it was this software that we choose in order to develop a simulation of ROMEO. As the people at LAGADIC had some experience with this software, they advised me to choose this software so that they were able to help me with the different problems that I encountered.



```
Selected objects:                      0
Simulation time:                       00:00:12.45 (dt=50.0 ms)
Script(s) executed                     Non-threaded: 1 (20 ms), running threads: 0 (0 ms)
Collision handling enabled             Calculations: 0, detections: 0 (0 ms)
Distance handling enabled              Calculations: 0 (0 ms)
Proximity sensor handling enabled      Calculations: 0, detections: 0 (0 ms)
Vision sensor handling enabled (FBO)   Calculations: 0, detections: 0 (0 ms)
IK group handling enabled              Calculations: 0 (0 ms)
Mechanism handling enabled             Calculations: 0 (0 ms)
Dynamics handling enabled (Bullet)     Calculation passes: 10 (15 ms)
Mill handling enabled                  Calculations: 0, surface cut: 0 mm^2 (0 ms)
```

ILLUSTRATION 12: V-REP WITH A NAO ROBOT

II. Missions

1. Simulation of ROMEO

When I arrived in the LAGADIC team, ROMEO was already able to grab a box on a table and to guide a marble in a labyrinth thanks to its vision. Thus, the first objective given to me was to create a simulation environment in which we could reproduce these demonstrations without physically using the robot. For this, the idea was to start from a simulation created for NAO to adapt it to ROMEO, which has many similarities with its little brother, except for the difference in size. A brief overview of the different characteristics of NAO was made in order to allow the comparison with ROMEO.

NAO is a small robot made of 26 joints and 24 different sensors (Illustration 13), including sonars, contact and tactile sensors, and some force sensitive resistors, plus 1 inertial unit. To see, it has two cameras, one on its forehead and another in its mouth. And to complement this vision, NAO is equipped with two other infrared sensor that are located in its eyes. All this technology is compressed in a body that is 57cm tall, and that weight only 5.4kg. Thanks to all its sensors and joints, NAO represent a perfect platform for discovering the joy of working with robots, even for young children, as they can easily program NAO with the Choregraphe software that is given with the robot. This is why this robot is very often seen in many classrooms, and we can find a simulated version of this robot on almost all the simulations software that are available, either with Gazebo, or Webots, or V-REP. The last one being the simulation software that I used.

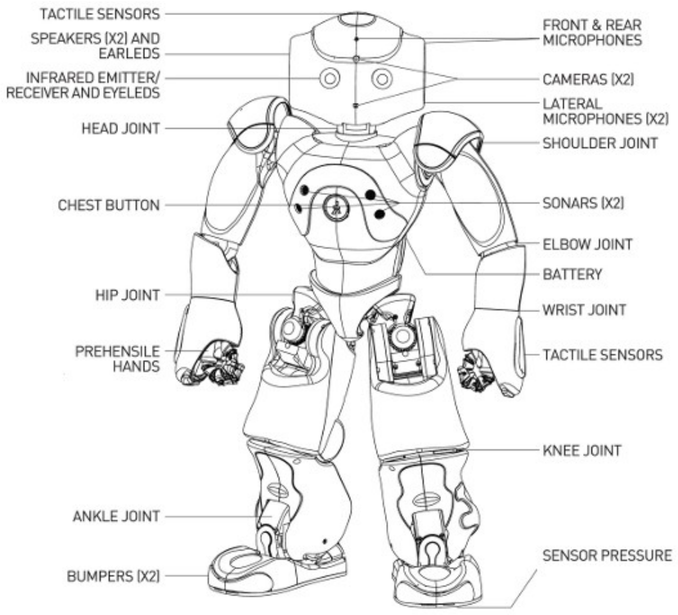


ILLUSTRATION 13: NAO'S HARDWARE

To begin with, I used a plugin that was developed by Marco Cognetti, an italian researcher in Rome (Italy), that created a bridge between Choregraphe and V-REP. The plugin was using a simulator-SDK given by Softbank Robotics that was interacting with the robot simulated in V-REP (Illustration 14), and was able to give the feedback to the artificial intelligence, that commands the robot, which is the NAOqi Firmware, also developed by Softbank Robotics. And finally, with Choregraphe, we could give orders to NAOqi, which was transmitting them to the simulator-SDK which in turns gave them to V-REP.

One of the arduous part of this plugin was to connect all the joints of NAO to its simulated counterpart. But thanks to a script directly implemented on the NAO model in V-REP, this connection was realized. (see code in annex 4).

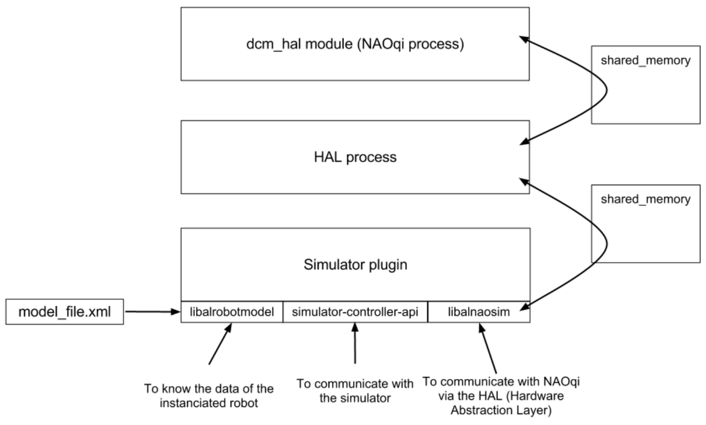

dcm_hal module (NAOqi process)
shared_memory
HAL process
shared_memory
Simulator plugin
model_file.xml
libalrobotmodel
simulator-controller-api
libalnaosim
To know the data of the instanciated robot
To communicate with the simulator
To communicate with NAOqi via the HAL (Hardware Abstraction Layer)

ILLUSTRATION 14: COMMUNICATION BETWEEN SIMULATOR-SDK AND NAOQI

After a few weeks of familiarization with the software and the plugin, we succeeded with Giovanni to control a simulated ROMEO through the Choregraphe software (Illustration 15). However, we could still not control the motion of the hands of the robot, an essential task when it comes to catching an object. Afterwards, we noticed that we were not working with the latest version of the robot control.

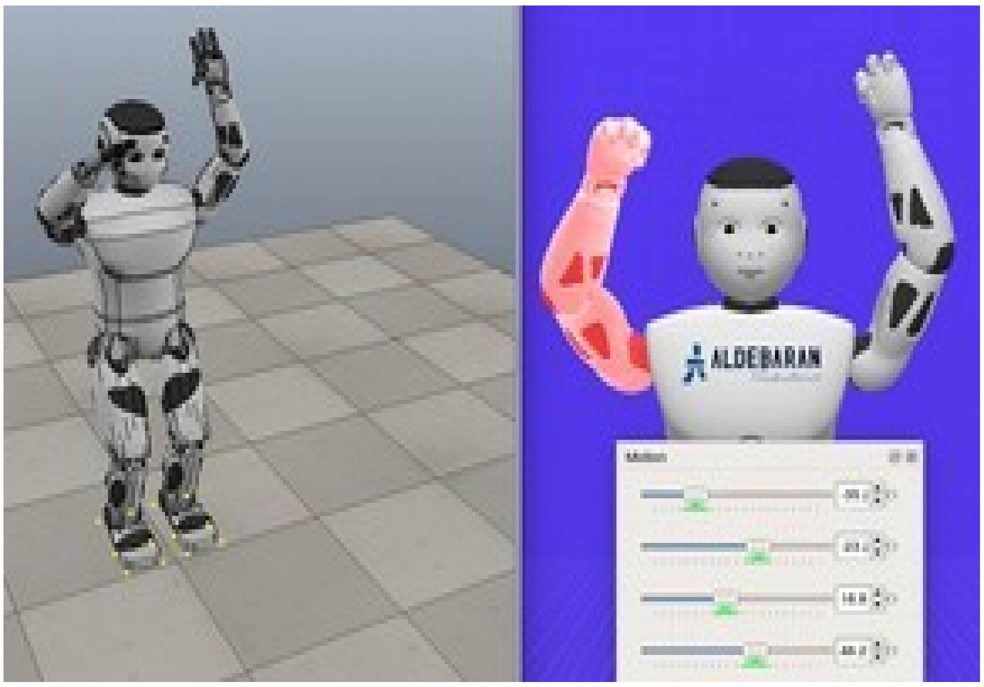


ILLUSTRATION 15: SIMULATED ROMEO CONTROLLED WITH CHOREGRAPHE

It was as that time that we ran into a compatibility problem. We had the possibility to use the last version of the robot controller : a NAOqi 2.3, but no simulator-SDK existed to communicate with this last version. After deferring this problem to the designers of ROMEO (Softbank Robotics Europe), we understood from their answer that we would have to wait several weeks without being able to advance on the simulation, and so, we decided to change objectives, and to start what constituted the major part of my internship – the realization of a program to open and close a door with ROMEO.

2. Using a door with ROMEO

In order to improve the functionalities of ROMEO, Fabien and Giovanni already had the idea to make it able to use a door, which basically means that it have to action the door handle. And in order to do that, ROMEO had to be able to obtain the exact localization of the door handle before moving its arm and actioning it.

We decided not to try to work with the two fixed cameras that are on the head of ROMEO, because the LAGADIC team already tried to work with stereo vision and had a lot of problems that did not exist with an RGB-D camera. The next section will explain how a RGB-D camera works. I spent some time looking at the different functions, plugins, and programs that already existed for RGB-D cameras, and apart from a few programs that detected different objects on a table, for example, a can of coke, or a square box, we found nothing that satisfied the team, and that could satisfy our needs.

Which is why we decided to start from scratch the program that would allows us to localize a door handle in a 3D environment. After the localization of the door handle, the next step was to use the previous functions developed by Fabien, Giovanni, and Don Jovan to make ROMEO move its arm in order to make it able to use the door - opening and closing it.

III. Localization of a door handle

1. Localization

To localize a door handle, a RGB-D camera was used, the Intel SR300 camera, which we mounted on ROMEO's skull, despite the fact that he already has four RGB cameras (Illustration 16) on his head. The two cameras at the level of its eyes, which can move according to the movements of the eyes and the two cameras on its forehead were not useful to obtain the localization of the door handle.

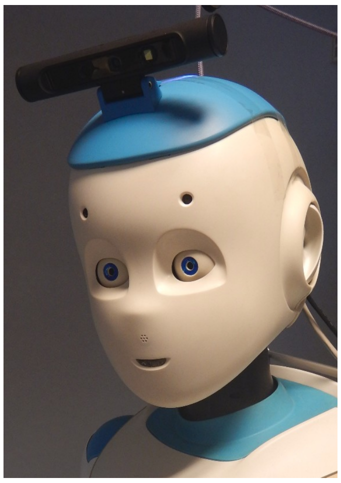


ILLUSTRATION 16: RGB-D CAMERA ON TOP OF ROMEO’S HEAD

Using the RGB-D camera, we obtain three different images: a RGB image (Illustration 17), an infrared image (Illustration 18) and a depth image (Illustration 19). These three images can be combined to obtain a point cloud (Illustration 20), which is the basic structure that will be used to localize the door handle. (See code in annex 5).

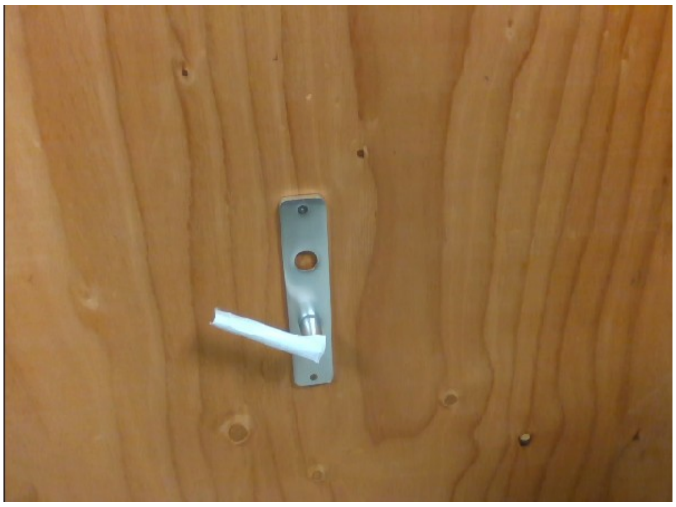


ILLUSTRATION 17: RGB IMAGE

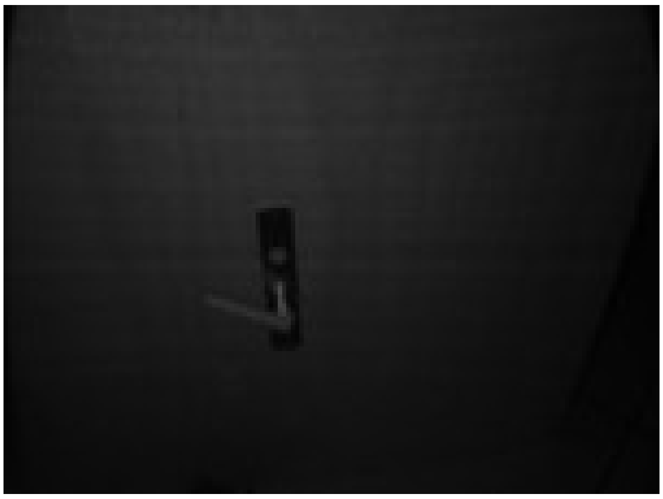


ILLUSTRATION 18: INFRARED IMAGE

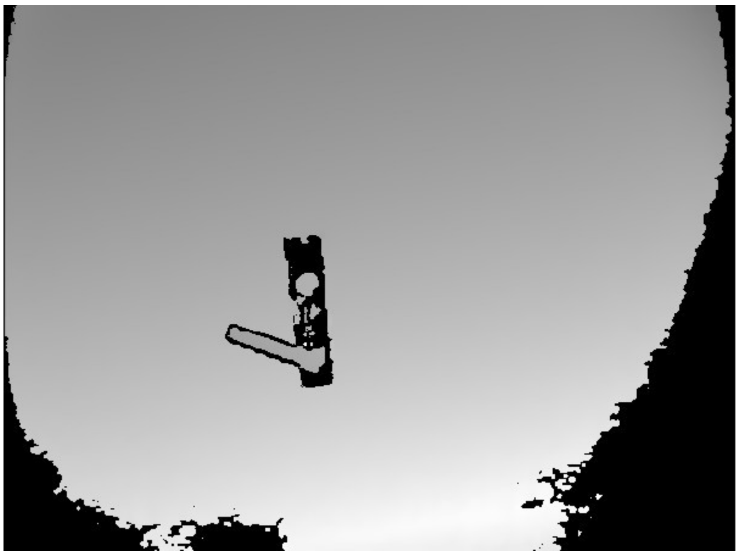


ILLUSTRATION 19: DEPTH IMAGE

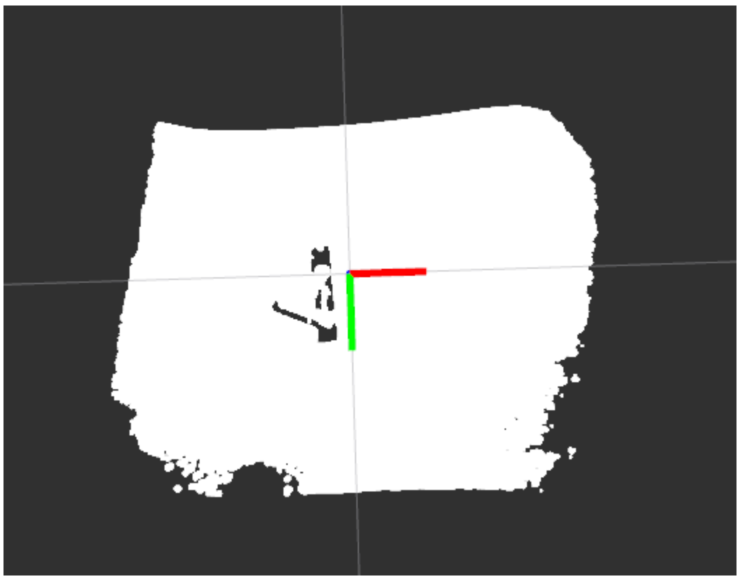


ILLUSTRATION 20: POINT CLOUD

1.1. Detection of the plane of a door

To estimate the localization of a door handle, we first had to find the plane of a door. For this, I used the segmentation provided by the Point Cloud Library, as explained in the section I.2.2. I thus obtained an equation (Eq.1) of the normal to the plane and the inliers of the said plane (Illustration 21 and 22). As this normal is used later for the localization of the handle, it will be called n_p (Eq.2).

$$\text{Equation of a plane : } aX + bY + cZ = -d \quad \textit{(Eq. 1)}$$

$$n_p = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad \textit{(Eq. 2)}$$

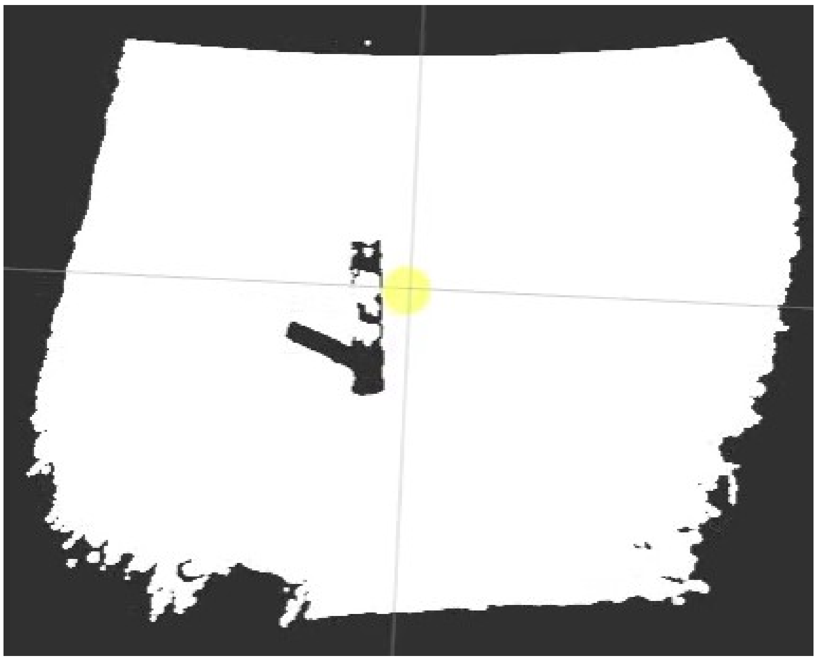


ILLUSTRATION 21: VIEW FROM THE FRONT

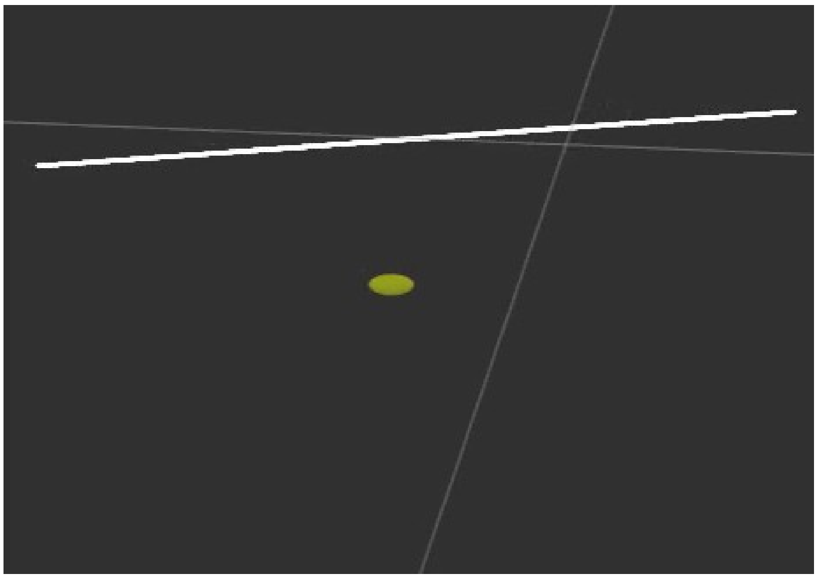


ILLUSTRATION 22: VIEW FROM THE TOP

1.2. Estimation of the door handle

From the plane of the door, the extraction of the points belonging almost only on the door handle was possible. In order to do that, I selected all the points that were located between two other planes parallel to the one of the door, respectively 5cm and 9cm in front of the door's plane. Those values were chosen as a door handle is usually placed at this distance from the door. From our observations, the points thus extracted belonged mostly to the handle of the door. (Illustration 23) (See code in annex 6)

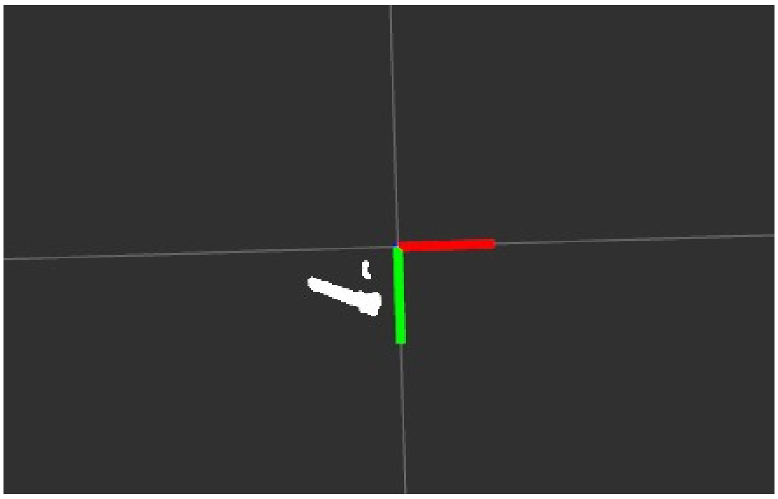


ILLUSTRATION 23: POINTCLOUD OF THE DOOR HANDLE

To localize the axis of the door handle in space, an orthogonal distance regression method was used to obtain the equation of the handle vector. What was desired was to get a right-hand equation of this form (Eq.3) :

$$x=x_0+a_l t$$
$$y=y_0+b_l t$$
$$z=z_0+c_l t$$

(Eq. 3)

The aim of this method is to minimize the sum of the squared distances between each point and the line. The vector equation of this sum leads to the following function (Eq.4).

$$f(x_0,y_0,z_0,a_l,b_l,c_l)=\sum(c_l(y_i-y_0)-b_l(z-z0))^2+(a_l(z_i-z_0)-c_l(x-x_0))^2+(b_l(x_i-x_0)-a_l(y-y_0))^2$$

(Eq.4)

Taking the first derivative with respect to x_0, y_0, and z_0 and setting the result to zero, we get equations that can be manipulated to get this equation (Eq.5) :

$$\frac{(x_0-\dot{x})}{a_l}=\frac{(y_0-\dot{y})}{b_l}=\frac{(z_0-\dot{z})}{c_l}$$

(Eq. 5)

Where $\dot{x}$, $\dot{y}$, $\dot{z}$ correspond to the coordinates of the centroid of the given points.

By minimizing the following sum (Eq. 6), the coefficients (a_l, b_l, c_l) were obtained.

$$\sum(|X_i, L|)^2 = \sum(|X_i, C|)^2 - \sum(|X_i, P|)^2$$

(Eq. 6)

Where L is the calculated line, C is the barycenter, P the plane passing through C in such a way that L is perpendicular to P, and the different Xi being the different points extracted.

Thus, X_1 the directing vector of the line (Eq.7) is obtained and so is X_0 the centroid of the points (Eq.8), which is also the center of the handle.

$$\boldsymbol{X_l} = \begin{bmatrix} a_l \\ b_l \\ c_l \end{bmatrix} \text{ (Eq. 7) et } \boldsymbol{X_0} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} \text{ (Eq. 8).}$$

After obtaining this direction vector, we were able to use the normal of the plane of the door n_p that we had previously found, and with a cross product between this normal and the direction vector of the handle, we got the last vector that allows us to localize the handle relative to the depth camera. (Illustration 24) (See code in annex 7).

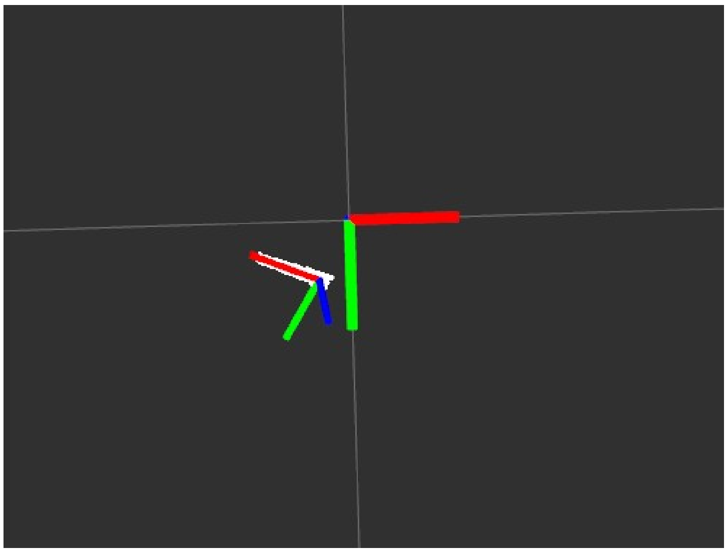


ILLUSTRATION 24: AXIS OF THE DOOR HANDLE

In order to be able to project this pose into the RGB image, we needed to use the extrinsic parameters that exist between the RGB camera and the depth camera. Because the sensors can not be located exactly in the same place, and they are separated by approximately 2.5cm. (Eq.9)

$${}^{c}M_h = ({}^{d}M_c)^{-1} . {}^{d}M_h$$

(Eq.9)

Where c is the RGB camera, d is the depth camera, and h is the handle.

Thus, we are able to localize the handle directly in the RGB image.

The problem we were now facing was that it could only detect the handle but could not be disturbed by a hand approaching the handle, which ROMEO will necessarily do when he wants to action the handle. So we needed to find a method to reduce the region of interest for the localization of the handle.

2. Perturbations removal

To remove the perturbations, a black and white image was created where all the points that belonged to the handle, that is to say those that are between 5cm and 9cm of the door, are colored in white. Once this image was obtained, I had to use several erosions and dilations to get a sufficiently full and clean shape in order to be able to follow it. With the ViSP software, described in section I.2.1, I was then able to follow this blob and create a bounding box around it. (Illustration 25) (See code in annex 8).

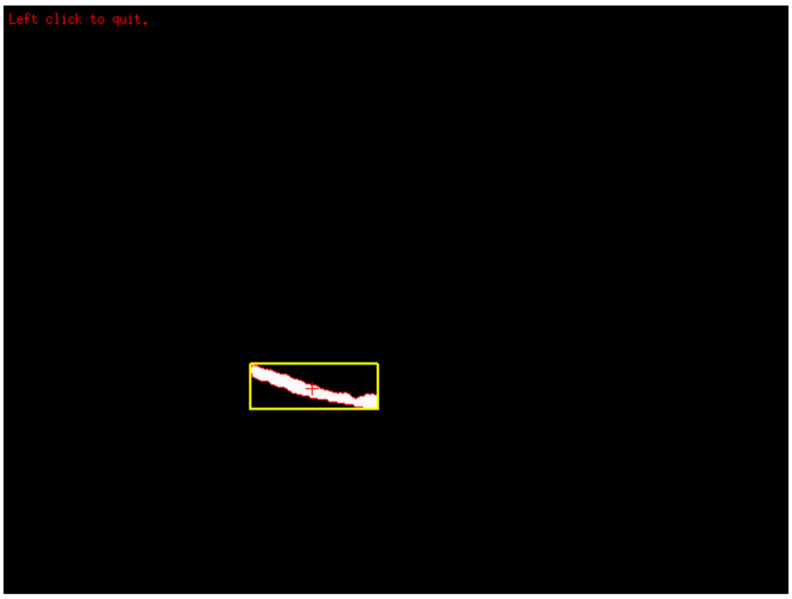


ILLUSTRATION 25: 2D IMAGE OF THE DOOR HANDLE

In this way, I only had to localize the door handle that was inside this box framing. Thus, even if someone approached another object that could look like a door handle, it was not detected and did not interfere with the detection.

The disadvantage of this method is that it concentrated the detection of the handle on the first object that looked like a handle. To overcome this problem, I added a possibility to click directly on the image to choose the blob that the program was designed to follow. This click is also useful when the tracking of the blob loses its target for any reason (sudden difference in luminosity, object passing in front of the handle, movement of the head of ROMEO too sudden, etc.).

Now that the handle was correctly localized, and was no more disturbed by any approaching object, I was able to concentrate my research on the actions that ROMEO had to make in order to open and close a door.

IV. Using a door with ROMEO

To use a door, ROMEO needed several ROS nodes that worked at the same time. He needed the localization of the door handle, but also the exact localization of its own hand. And this last part is achieved with its vision, because the different joints of its arm are not extremely rigid, and a localization by odometry would not be precise enough. To localize its own hand, I used a program that was already created by Giovanni which follows a target composed of four blobs, that was fixed on ROMEO's wrist, which gives the pose of the hand of ROMEO. (Illustration 26)

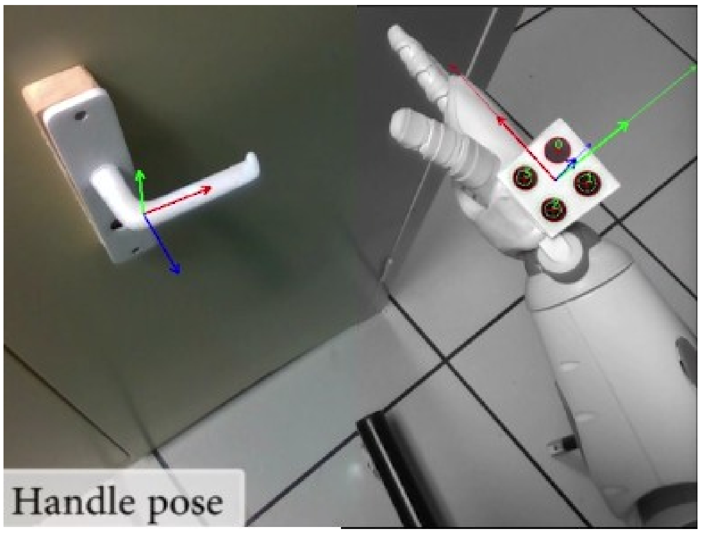


ILLUSTRATION 26: POSES OF THE DOOR HANDLE AND THE HAND OF ROMEO

Then, with these two poses, I could find a way to move ROMEO's arm in order to make it put its hand firstly near the door handle and then grab it in order to open and pull the door towards the robot.

The movement needed to be broken down in seven parts, each of these realized in a different way.

Firstly, with Giovanni, we used an open-loop movement for the arm of ROMEO in order to see its hand and the handle of the door in the same image from a forehead camera. An open-loop movement means that the robot will execute the movement and not take into account any output that it could receive. This method is quite crude and not very robust, but as we were nearing the end of my internship, we had to make a demonstration for the other people of the lab, which is why we choose this solution. In order to do this, we used the Cartesian coordinates of a point to be reached by its hand, that we saved when the wrist of ROMEO and the door handle were both visible by the 2D camera that is on ROMEO's right eye. Then, a program developed by Giovanni before my arrival in the team could transform the coordinates into a speed command for each of the engines of ROMEO's arm. (see code in annex 9).

The second part is a closed-loop movement, realized with a PBVS - Pose Based Visual Servoing. This servoing makes it possible to calculate the speeds to be transmitted to the various joints of ROMEO's arm so that from a current position (the pose of the hand of ROMEO), it arrives at a desired position. As it is a closed-loop movement, the execution of this movement is robust and can compensate any offset that could exist because of the first open-loop movement. We took into account the fact that the desired position is not exactly the pose of the handle, but rather the handle with a slight offset. Indeed, if ROMEO covers the handle of his hand, the localization of the handle could no longer be done. This is why we added a constant offset to the localization of the handle. Thus, we were able to position the hand of ROMEO close enough to the handle so that it only had to make a small movement in order to catch the handle, while keeping it far enough not to disturb the localization of the handle. (Illustration 27) (See code in annex 10).

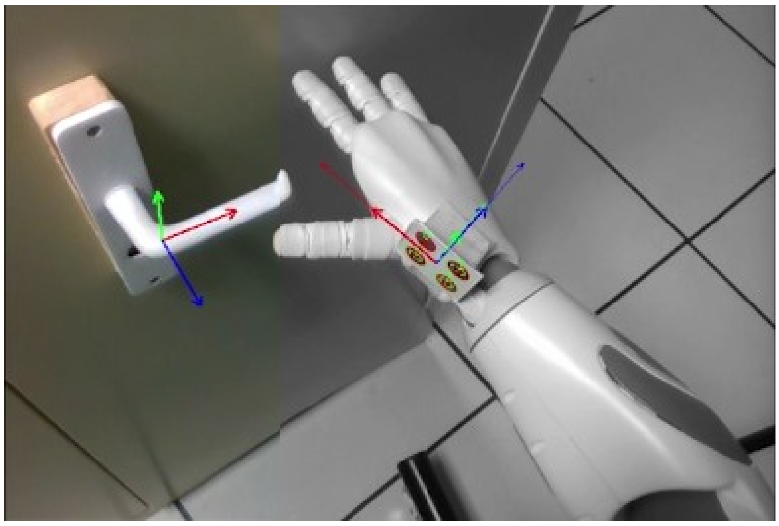


ILLUSTRATION 27: THE HAND OF ROMEO IS SERVOED TO THE POSE OF THE DOOR HANDLE

The third movement is another open-loop movement where the hand of ROMEO is directed along the three-dimensions axes located at the level of his wrist. It is thus possible to make the hand of ROMEO perform rotations and translations. This method, although practical, can prove itself dangerous if the robot arm finds itself in a position of singularity. We were able to experience it ourselves. With this third movement, ROMEO's hand was therefore placed just above the handle. (Illustration 28) (See code in annex 11).

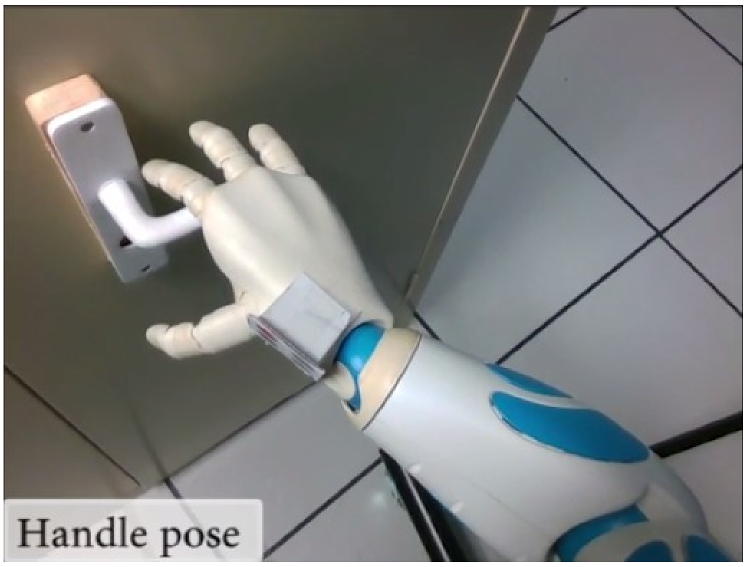


ILLUSTRATION 28: HAND OF ROMEO PLACED ON TOP OF DOOR HANDLE

For the fourth movement, we had to make the hand of ROMEO close around the handle, while making it move forward towards the door. The aim of this movement was to prevent its fingers from stopping against the door while trying to grab as much as possible the handle with the palm of ROMEO and not just its fingers. At the beginning, we were just closing ROMEO's hand, but we then realized that when ROMEO wanted to rotate the door handle, its fingers' strength was not enough to keep the door handle in its hand. This problem was due to a hardware safety measure, because as the fingers of ROMEO are cable-linked, the motor for closing the hand had a limited output in order not to break the cable, and risking damaging the robot. (Illustration 29) (See code in annex 12).

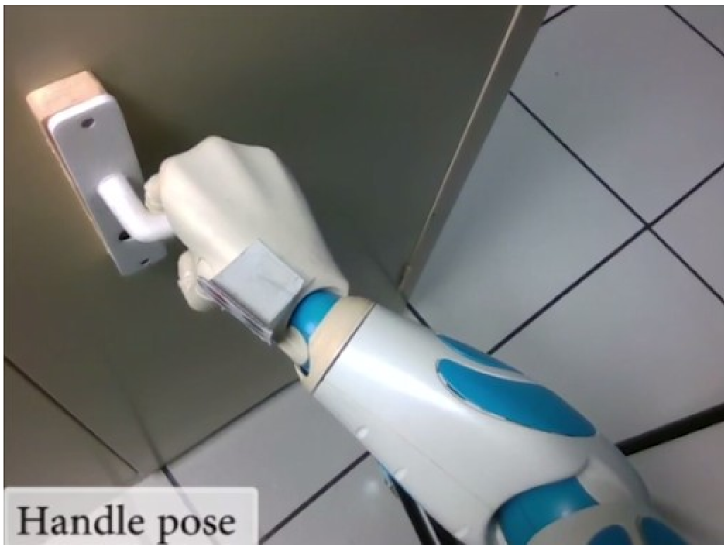


ILLUSTRATION 29: DOOR HANDLE GRABBED

After having grabbed the door handle, ROMEO had to lower his hand while turning its wrist so that the handle of the door rotates and that the mechanism blocking the door unlocks. For this movement again, we used an open loop movement, as there was no reference point that could have allowed us to perform a more robust closed-loop movement. (Illustration 30) (See code in annex 13).

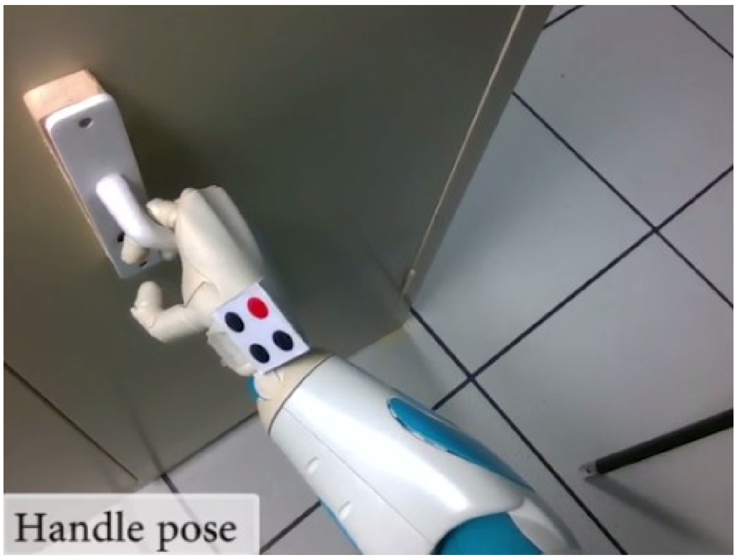


ILLUSTRATION 30: ACTIONNING THE DOOR HANDLE

After ROMEO unlocked the door, by rotating the door handle, we had to make it pull the door towards it. To realize this movement, we once again relied on an open-loop movement, where ROMEO pull back its hand towards its body (along the x-axis on its wrist). (Illustration 31) (See code in annex 14).

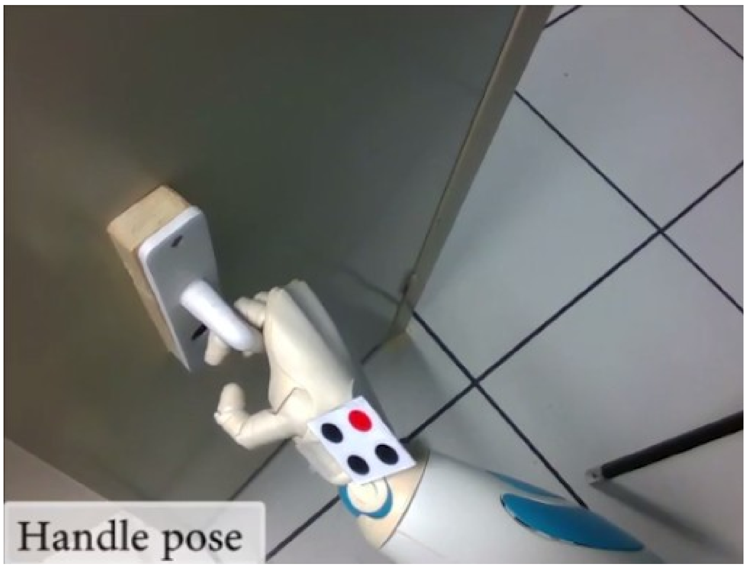


ILLUSTRATION 31: DOOR OPENED

To finish the demonstration, we had to make ROMEO releases the handle and makes it spread its arm in order to allow the person who had asked it to open the door to cross it (if it is a door between two rooms) or to fetch something in the cabinet which ROMEO has just opened. (Illustration 32) (See code in annex 15).

ILLUSTRATION 32: ROMEO'S HAND AWAY FROM DOOR HANDLE

As we were finishing the programming of the different movements to open the door, we took the time to realize the movements needed to close a door, as only a few movements had to be reversed, but the main structure of the code could be re-used. (See code in annex 16).

Conclusion

After five months of work with the LAGADIC team, I am proud to have succeeded in carrying out this project. During this internship, I was able to put into practice my knowledge acquired both in France and Italy. This internship allowed me to realize a real first project, with the realization at the end of a video demonstration.

The objective is fully realized, with both the robust localization of a door handle and the possibility for ROMEO to open a door.

Thanks to this internship, I learned a lot, both in the field of vision, and in programming that was only done in C ++, which I discovered, thanks to the advices of Fabien and Giovanni.

I encountered several difficulties during this internship, especially when reducing the region of interest of the image for the localization of the handle. But after many tests and discussions with Fabien and Giovanni, we finally came up with a satisfying result. This allowed me to question my work and to think about all the possible solutions to a problem before trying them, thus developing my rigor and concentration.

I really enjoyed doing a project as a whole and visualizing what each step required as work, to interact with the other members of the team to validate certain steps. I discovered with passion the field of humanoid robotic and acquired a good knowledge thanks to this project. It might be interesting to rework certain parts of ROMEO's movements in order to make it more robust or to improve the localization of the handle in order to be able to detect and locate handles of different shapes.

Another idea that we discussed with Giovanni but that we could not realize was to implement this program on PEPPER, another robot of Softbank Robotics. We would have been faced with new problems as PEPPER is smaller than ROMEO, and possess less degrees of freedom in its arms, but it would have been interesting to see if the work that was done for ROMEO was easily adaptable for PEPPER.

Bibliography

Giovanni Claudio, Fabien Spindler and François Chaumette (2016) Vision-based manipulation with the humanoid robot ROMEO

Fernandez, E., Crespo, L. S., Mahtani, A., & Martinez, A. (2015). Learning ROS for Robotics Programming. Packt Publishing Ltd.

Ros.org (http://wiki.ros.org/) a website to get started with ROS

V-REP Software (http://www.coppeliarobotics.com/helpFiles/index.html) the documentation of the V-REP simulator

ViSP (http://visp-doc.inria.fr/doxygen/visp-daily/index.html/) the documentation of the vision software developed by the LAGADIC Team

PCL (http://pointclouds.org/documentation/) the documentation of the PCL software

Wikipedia (http://wikipedia.org/) Free encyclopedia

Video demonstration of the realized project (https://www.youtube.com/watch?v=qotsmwXmTUY)

Source code of the realized project (https://github.com/lagadic/door_handle_detection & https://github.com/lagadic/demo_romeo_door)

Annexes

Annexes' Index

ANNEX 1. Region Of Interest Filter

```
pcl::PointCloud<pcl::PointXYZ>::Ptr    DoorHandleDetectionNode::getOnlyUsefulHandle(const
pcl::PointCloud<pcl::PointXYZ>::Ptr &cloud)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_useful(new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered(new pcl::PointCloud<pcl::PointXYZ>);
[...]
  //Use the passthrough filter to elimate all the point that are not inside the limits
  pcl::PassThrough<pcl::PointXYZ> pass1;
  pass1.setInputCloud (cloud);
  pass1.setFilterFieldName ("x");
  pass1.setFilterLimits (m_X_min, m_X_max);
  pass1.filter (*cloud_filtered);

  pcl::PassThrough<pcl::PointXYZ> pass2;
  pass2.setInputCloud (cloud_filtered);
  pass2.setFilterFieldName ("y");
  pass2.setFilterLimits (m_Y_min, m_Y_max);
  pass2.filter (*cloud_useful);

  return cloud_useful;
}
```

ANNEX 2. Voxel Grid Filter

```
void  DoorHandleDetectionNode::mainComputation(const  sensor_msgs::PointCloud2::ConstPtr
&image)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);
[...]
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered(new pcl::PointCloud<pcl::PointXYZ>);

  //Downsample the point cloud given by the camera
  pcl::VoxelGrid<pcl::PointXYZ> vg;
  vg.setInputCloud( cloud );
  vg.setLeafSize( 0.005f, 0.005f, 0.005f );
  vg.filter(*cloud_filtered);
[...]
}
```

ANNEX 3. Plane Segmentation

```
inliersAndCoefficients DoorHandleDetectionNode::getPlaneInliersAndCoefficients(const
pcl::PointCloud<pcl::PointXYZ>::Ptr &cloud)
{
  struct inliersAndCoefficients plane;
  pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients ());
  pcl::PointIndices::Ptr inliers(new pcl::PointIndices);
  Eigen::Vector3f axis_plane(0., -0.5, 0.5);
[...]
   //Use a plane segmentation to find the inliers and coefficients for the biggest plan
found near the z axis
  pcl::SACSegmentation<pcl::PointXYZ> seg;
  seg.setOptimizeCoefficients (true);
  seg.setModelType (pcl::SACMODEL_PLANE);
  seg.setMethodType (pcl::SAC_RANSAC);
  seg.setAxis( axis_plane );
  seg.setDistanceThreshold (0.02);
  seg.setInputCloud (cloud);
  seg.segment (*inliers, *coefficients);

  plane.coefficients = coefficients;
  plane.inliers = inliers;
[...]
  return plane;
}
```

ANNEX 4. Bridge script between V-REP and Choregraphe

```
if (sim_call_type==sim_childscriptcall_initialization) then

    -- Determin the colors we want
    whiteStr=simGetScriptSimulationParameter(sim_handle_self,'whiteColor')
    greyStr=simGetScriptSimulationParameter(sim_handle_self,'greyColor')
    w={0.46,0.46,0.46}
    g={0.29,0.29,0.29}
    i=1
    for token in string.gmatch(whiteStr,"[^%s]+") do
        w[i]=token
        i=i+1
    end
    i=1
    for token in string.gmatch(greyStr,"[^%s]+") do
        g[i]=token
        i=i+1
    end
    -- Get all the visible shapes in this model:
    allObjectsToExplore={simGetObjectAssociatedWithScript(sim_handle_self)}
    allVisibleShapes={}
    while (#allObjectsToExplore>0) do
        obj=allObjectsToExplore[1]
        table.remove(allObjectsToExplore,1)
        if (simGetObjectType(obj)==sim_object_shape_type) then
              r,v=simGetObjectInt32Parameter(obj,sim_objintparam_visibility_layer) -- get
the layers this shape is visible in
             if (v<256) then -- by default, the first 8 layers are visible, the last ones
are invisible
                table.insert(allVisibleShapes,obj)
            end
        end
        index=0
        while true do
            child=simGetObjectChild(obj,index)
            if (child==-1) then
                break
            end
            table.insert(allObjectsToExplore,child)
            index=index+1
        end
    end
    -- Now change the color of all those shapes:
    for i=1,#allVisibleShapes,1 do
        simSetShapeColor(colorCorrectionFunction(allVisibleShapes[i]),'NAO_WHITE',0,w)
        simSetShapeColor(colorCorrectionFunction(allVisibleShapes[i]),'NAO_GREY',0,g)
    end
end
```

ANNEX 5. Recuperation of the PointCloud

```
DoorHandleDetectionNode::DoorHandleDetectionNode(ros::NodeHandle nh)
{
[...]
    pcl_frame_sub  =  n.subscribe(  m_pclTopicName,  1,  (boost::function  <  void(const
sensor_msgs::PointCloud2::ConstPtr&)>)
boost::bind( &DoorHandleDetectionNode::mainComputation, this, _1 ));
[...]
}

void  DoorHandleDetectionNode::mainComputation(const  sensor_msgs::PointCloud2::ConstPtr
&image)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);
[...]
  //Convert the sensor_msgs/PointCloud2 data to pcl/PointCloud
  pcl::fromROSMsg (*image, *cloud);
[...]
}
```

ANNEX 6. PointCloud with only the door handle

```
pcl::PointCloud<pcl::PointXYZ>::Ptr      DoorHandleDetectionNode::createPCLSandwich(const
pcl::PointCloud<pcl::PointXYZ>::Ptr & cloud, vpColVector coefficients)

{

  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_outside(new pcl::PointCloud<pcl::PointXYZ>);
  cloud_outside->width = 300000;
  cloud_outside->height = 1;
  cloud_outside->points.resize (cloud_outside->width * cloud_outside->height);

  double xc, yc, zc, z_min, z_max;
  size_t width_outside = 0;

  for(int i = 0; i < cloud->size(); i++)
  {
    if (cloud->points[i].z !=0)
    {
      xc = cloud->points[i].x;
      yc = cloud->points[i].y;
      zc = cloud->points[i].z;

      //Create a zmin and zmax for every point to check if the point is outside or inside
the detection
       z_min = -(coefficients[0]*xc + coefficients[1]*yc + (coefficients[3] + m_height_dh
+ 0.02) )/(coefficients[2]);
       z_max = -(coefficients[0]*xc + coefficients[1]*yc + (coefficients[3] + m_height_dh
- 0.02) )/(coefficients[2]);

      //If the point is inside, we add it to the new point cloud
      if (cloud->points[i].z > z_min && cloud->points[i].z < z_max )
      {
        width_outside++;
        cloud_outside->points[width_outside-1].x = xc;
        cloud_outside->points[width_outside-1].y = yc;
        cloud_outside->points[width_outside-1].z = zc;
      }
    }
  }
  cloud_outside->width = width_outside;
  cloud_outside->points.resize (cloud_outside->width * cloud_outside->height);

  return cloud_outside;
}
```

ANNEX 7. Axis of the door handle

```
vpColVector DoorHandleDetectionNode::getCoeffLineWithODR(const
pcl::PointCloud<pcl::PointXYZ>::Ptr &cloud)
{
  vpMatrix M(cloud->size(),3);
  vpRowVector m(3);
  vpColVector centroid = DoorHandleDetectionNode::getCentroidPCL(cloud);

  //Create a Matrix(n,3) with the coordinates of all the points
  for(unsigned int i=0; i<cloud->size(); i++) {
    m[0] = cloud->points[i].x - centroid[0];
    m[1] = cloud->points[i].y - centroid[1];
    m[2] = cloud->points[i].z - centroid[2];
    for(unsigned int j=0;j<3;j++)
      M[i][j] = m[j];
  }

  vpMatrix A = M.t() * M;

  vpColVector D;
  vpMatrix V;
  A.svd(D, V);

  double largestSv = D[0];
  unsigned int indexLargestSv = 0 ;
  for (unsigned int i = 1; i < D.size(); i++) {
    if ((D[i] > largestSv) ) {
      largestSv = D[i];
      indexLargestSv = i;
    }
  }

  vpColVector h = V.getCol(indexLargestSv);

  return h;
}
```

ANNEX 8. Conversion of the image in 2D and bounding box

```
void  DoorHandleDetectionNode::morphoSandwich(const  pcl::PointCloud<pcl::PointXYZ>::Ptr
&cloud) {

  m_img_mono = 0;
  double X,Y,Z,x,y,u,v;
  vpImagePoint bottomRightBBoxHandle;
  vpImagePoint topLeftBBoxHandle;
  vpMouseButton::vpMouseButtonType button;
  vpRect bboxhandle, searchingField;

  //Convert the 3D points in 2D
  for(int i = 0; i < cloud->size(); i++ ){
    X = cloud->points[i].x ;
    Y = cloud->points[i].y ;
    Z = cloud->points[i].z ;
    x = X/Z;
    y = Y/Z;

    //Convert the points in meters to points in pixels
    vpMeterPixelConversion::convertPoint(m_cam_depth, x, y, u, v);

    //Color theses points in white
    if(u < m_img_mono.getWidth()-1 && v < m_img_mono.getHeight()-1 && u > 0 && v > 0 )
      m_img_mono.bitmap[ (int)v * m_img_mono.getWidth() + (int)u ] = 250;
  }

  //Use some dilatation/erosion to have a blob that is easy to track
   vpImageMorphology::dilatation(m_img_mono,  (unsigned  char)  250,  (unsigned  char)  0,
vpImageMorphology::CONNEXITY_4);
   vpImageMorphology::dilatation(m_img_mono,  (unsigned  char)  250,  (unsigned  char)  0,
vpImageMorphology::CONNEXITY_4);
    vpImageMorphology::erosion(m_img_mono,  (unsigned  char)  250,  (unsigned  char)  0,
vpImageMorphology::CONNEXITY_4);
    vpImageMorphology::erosion(m_img_mono,  (unsigned  char)  250,  (unsigned  char)  0,
vpImageMorphology::CONNEXITY_4);
    vpImageMorphology::erosion(m_img_mono,  (unsigned  char)  250,  (unsigned  char)  0,
vpImageMorphology::CONNEXITY_4);
   vpImageMorphology::dilatation(m_img_mono,  (unsigned  char)  250,  (unsigned  char)  0,
vpImageMorphology::CONNEXITY_4);

  //Display and track the white blob inside the image
  vpDisplay::display(m_img_mono);

  if ( !m_tracking_is_initialized )
  {
        if  (  m_pointPoseHandle.get_u()  >  0  &&  m_pointPoseHandle.get_v()  >  0  &&
m_pointPoseHandle.get_u()  <  m_img_mono.getWidth()-1  &&  m_pointPoseHandle.get_v()  <
m_img_mono.getHeight()-1 )
    {
      m_blob.initTracking(m_img_mono, m_pointPoseHandle, 150, 255);
      m_tracking_is_initialized = true;
    }
  }
  else
  {
    m_blob.track(m_img_mono);
    bboxhandle = m_blob.getBBox();
    m_tracking_works = true;
  }
}
```

ANNEX 9. Moving the head and the arm towards the door handle

```
if (state == HeadToZero)    {

      //OpenLoop Control to see the door handle
      head_pose = 0;
      head_pose[0] = vpMath::rad(-24.3); // NeckYaw
      head_pose[1] = vpMath::rad(16.2); // NeckPitch
      head_pose[2] = vpMath::rad(-7.6); // HeadPitch
      head_pose[3] = vpMath::rad(0.0); // HeadRoll
      romeo.setPosition(jointNamesHead, head_pose, 0.06);
      state = WaitHeadToZero;
           vpDisplay::displayText(img_,  vpImagePoint(15,10),  "Head  should  go  in  zero
position", vpColor::red);
      ROS_INFO("Head should go in zero position");

    }


void DemoRomeoDoor::moveRArmFromRestPosition ()
{

  try
  {
       AL::ALValue  pos1  =  AL::ALValue::array(0.3741794228553772,  -0.33311545848846436,
-0.036883752793073654, 1.260278582572937, 0.4322237968444824, 0.009434251114726067);
       AL::ALValue  pos2  =  AL::ALValue::array(0.39718443155288696,  -0.282814621925354,
0.17343932390213013, 1.194741129875183, -0.5093367099761963, 0.18652880191802979);
       AL::ALValue  pos3  =  AL::ALValue::array(0.3460591435432434,  -0.2849748432636261,
0.08855552971363068, 0.9453462958335876, -0.11968476325273514, 0.25619229674339294);

    AL::ALValue time1 = 1.5f;
    AL::ALValue time2 = 3.0f;
    AL::ALValue time3 = 5.0f;

    AL::ALValue path;
    path.arrayPush(pos1);
    path.arrayPush(pos2);
    path.arrayPush(pos3);

    AL::ALValue times;
    times.arrayPush(time1);
    times.arrayPush(time2);
    times.arrayPush(time3);

    AL::ALValue chainName  = AL::ALValue::array ("RArm");
    AL::ALValue space      = AL::ALValue::array (0); // Torso
    AL::ALValue axisMask   = AL::ALValue::array (63);

    romeo.getProxy()->positionInterpolations(chainName, space, path, axisMask, times);
  }
  catch(const std::exception&)
  {
    throw vpRobotException (vpRobotException::badValue,
                            "servo apply the motion");
  }

  return;
}
```

ANNEX 10. Pose Based Visual Servoing

```
void DemoRomeoDoor::computeControlLaw(const vpHomogeneousMatrix &doorhandleMoffset)
{
    vpHomogeneousMatrix currentFeature;
    vpHomogeneousMatrix cMhandle_des;
    vpRotationMatrix cRh;
    vpTranslationVector cTh;
    geometry_msgs::Pose cMh_msg;
    tf::Transform transformdh;
    static tf::TransformBroadcaster br;

//      std::cout  <<  cMh_isInitialized  <<  "    "  <<  cMdh_isInitialized   <<  "    "  <<
statusPoseHand << "  " <<  statusPoseDesired << "  " << start_pbvs << std::endl;
    if ( status_hand && status_door_handle && start_pbvs == 1)
    {
        static bool first_time = true;
        if (first_time) {
            std::cout << "-- Start visual servoing of the arm" << std::endl;
            servo_time_init = vpTime::measureTimeSecond();
            first_time = false;
        }
        vpAdaptiveGain lambda(1.5, 0.12, 8);
        servo_arm.setLambda(lambda);
        servo_arm.set_eJe(romeo.get_eJe(chain_name));
        currentFeature = doorhandleMoffset.inverse() * cMdh.inverse() * cMh;
        cMhandle_des = cMdh * doorhandleMoffset;

        ////Publish the TF BEGIN////
              transformdh.setOrigin(  tf::Vector3(cMhandle_des[0][3],  cMhandle_des[1][3],
cMhandle_des[2][3] ));
        cTh = cMhandle_des.getTranslationVector();
        cRh = cMhandle_des.getRotationMatrix();
//        intern_cMh = vpHomogeneousMatrix(cTh, cRh);
        cMh_msg = visp_bridge::toGeometryMsgsPose(vpHomogeneousMatrix(cTh, cRh));

        tf::Quaternion qdh;
        qdh.setX(cMh_msg.orientation.x);
        qdh.setY(cMh_msg.orientation.y);
        qdh.setZ(cMh_msg.orientation.z);
        qdh.setW(cMh_msg.orientation.w);

        transformdh.setRotation(qdh);
                    br.sendTransform(tf::StampedTransform(transformdh,   ros::Time::now(),
"SR300_rgb_optical_frame", "desired_pose_tf"));
        ////Publish the TF END////
        servo_arm.setCurrentFeature(currentFeature) ;
        // Create twist matrix from target Frame to Arm end-effector (WristPitch)
        vpVelocityTwistMatrix oVe_LArm(oMe_Arm);
        servo_arm.m_task.set_cVe(oVe_LArm);

        //Compute velocities PBVS task
                  q_dot   =   -   servo_arm.computeControlLaw(vpTime::measureTimeSecond()   -
servo_time_init);

        q = romeo.getPosition(jointNames_arm);
         q2_dot   = servo_arm.m_task.secondaryTaskJointLimitAvoidance(q, q_dot, jointMin,
jointMax);

//        vpDisplay::displayFrame(img_, cMh, )
        publishCmdVel(q_dot + q2_dot);

        vpTranslationVector t_error_grasp = currentFeature.getTranslationVector();
        vpRotationMatrix R_error_grasp;
        currentFeature.extract(R_error_grasp);
        vpThetaUVector tu_error_grasp;
        tu_error_grasp.buildFrom(R_error_grasp);
        double theta_error_grasp;
        vpColVector u_error_grasp;
        tu_error_grasp.extract(theta_error_grasp, u_error_grasp);
        double error_t_treshold = 0.001;

        init = false;

        if ( (sqrt(t_error_grasp.sumSquare()) < error_t_treshold) && (theta_error_grasp <
vpMath::rad(3)) )
        {
          pbvs_finished = true;
          vpColVector q_dot_zero(numJoints,0);
          publishCmdVel(q_dot_zero);
        }
        std::cout << "We are servoing the arm " << start_pbvs << std::endl;
    }
    else if (!init && start_pbvs == 0)
    {
      init = true;
      vpColVector q_dot_zero(numJoints,0);
      publishCmdVel(q_dot_zero);
      std::cout << "publishing just once" << std::endl;
    }
    else if ( start_pbvs == 1 &&( status_hand == 0 || status_door_handle == 0) )
    {
      vpColVector q_dot_zero(numJoints,0);
      publishCmdVel(q_dot_zero);
    }

}
```

ANNEX 11. Moving the hand on top of the door handle

```
if (state == PutHandOnDoorHandle1)
    {
        romeo.getProxy()->setStiffnesses("RHand", 1.0f);
        AL::ALValue angle = 0.70;
        romeo.getProxy()->setAngles("RHand", angle, 0.50);
        vpTime::sleepMs(100);

        //Open loop upward motion of the hand
        vpColVector cart_delta_pos(6, 0);
        cart_delta_pos[5] = vpMath::rad(-5);
        cart_delta_pos[1] = 0.10;
        cart_delta_pos[2] = -0.025;
        double delta_t = 3;


        static vpCartesianDisplacement moveCartesian;
        vpVelocityTwistMatrix V;
        if (moveCartesian.computeVelocity(romeo, cart_delta_pos, delta_t, "RArm", V)) {
                                         romeo.setVelocity(moveCartesian.getJointNames(),
moveCartesian.getJointVelocity());
          //        ROS_INFO("Right Arm should go in openLoop to open the door");
        }
        else
        {
          romeo.stop(moveCartesian.getJointNames());
          vpDisplay::displayText(img_, vpImagePoint(15,10), "Left click to grasp the door
handle", vpColor::red);
          if (click_done && button == vpMouseButton::button1 ) {
            state = GraspingDoorHandle;
            click_done = false;
            once = 0;
          }
          ROS_INFO("Right Arm should have put the hand on the door handle");
        }
    }
```

ANNEX 12. Grasping the door handle

```
if (state == GraspingDoorHandle)
{
  romeo.getProxy()->setStiffnesses("RHand", 1.0f);
  AL::ALValue angle = 0.30;
  romeo.getProxy()->setAngles("RHand", angle, 0.30);
  vpTime::sleepMs(100);
  state = PutTheHandCloser;
  ROS_INFO("Right Arm should have grasped the handle");
}
if (state == PutTheHandCloser)
{
  // Open loop motion of the hand to put the hand of Romeo nearer to the door
  vpColVector cart_delta_pos(6, 0);
  cart_delta_pos[0] = 0.02;
  double delta_t = 2;

  static vpCartesianDisplacement moveCartesian;
  vpVelocityTwistMatrix V;
  if (moveCartesian.computeVelocity(romeo, cart_delta_pos, delta_t, "RArm", V)) {
    romeo.setVelocity(moveCartesian.getJointNames(), moveCartesian.getJointVelocity());
//    ROS_INFO("Right Arm should go in openLoop to open the door");
  }
  else
  {
    romeo.stop(moveCartesian.getJointNames());
    state = GraspingDoorHandle2;
    ROS_INFO("Right Arm should have finished to open the door");
  }
}
```

ANNEX 13. Rotating the door handle

```
if (state == RotatingHandle)
{ // Open loop motion of the hand to rotate the handle
  vpColVector cart_delta_pos(6, 0);
  cart_delta_pos[3] = vpMath::rad(+40);
  cart_delta_pos[2] = -0.07;
  double delta_t = 3;

  static vpCartesianDisplacement moveCartesian;
  vpVelocityTwistMatrix V;
  if (moveCartesian.computeVelocity(romeo, cart_delta_pos, delta_t, "RArm", V)) {
    romeo.setVelocity(moveCartesian.getJointNames(), moveCartesian.getJointVelocity());
  }
  else
  {
    romeo.stop(jointNames_tot);
      vpDisplay::displayText(img_, vpImagePoint(15,10), "Left click to open the door",
vpColor::red);
    if (click_done && button == vpMouseButton::button1 ) {
      state = OpenDoor;
      click_done = false;
    }
    ROS_INFO("Right Arm should have opened the door");
  }
}
```

ANNEX 14.   Pulling the door towards ROMEO

```
if (state == OpenDoor)
    {
      // Open loop motion of the hand to open the door
      vpColVector cart_delta_pos(6, 0);
      cart_delta_pos[0] = -0.10;
      cart_delta_pos[4] = vpMath::rad(+15);
      double delta_t = 3;

      door_closed = false;

      static vpCartesianDisplacement moveCartesian;
      vpVelocityTwistMatrix V;
      if (moveCartesian.computeVelocity(romeo, cart_delta_pos, delta_t, "RArm", V)) {
                                           romeo.setVelocity(moveCartesian.getJointNames(),
moveCartesian.getJointVelocity());
      }
      else
      {
        romeo.stop(moveCartesian.getJointNames());
        vpDisplay::displayText(img_, vpImagePoint(15,10), "Left click to close the door",
vpColor::red);
          vpDisplay::displayText(img_, vpImagePoint(30,10), "Middle kick to release the
door", vpColor::red);
        if (click_done && button == vpMouseButton::button1 ) {
          state = CloseDoor;
          click_done = false;
        }
        if (click_done && button == vpMouseButton::button2 ) {
          state = ReleaseDoorHandle;
          click_done = false;
        }
        ROS_INFO("Right Arm should have finished to open the door");
      }
    }
```

ANNEX 15.   Releasing the door handle

```
if (state == ReleaseDoorHandle)
    {
      romeo.getProxy()->setStiffnesses("RHand", 1.0f);
      AL::ALValue angle = 0.5;
      romeo.getProxy()->setAngles("RHand", angle, 0.1);
      moveRArmToRestPosition();

      state = GoBacktoResPosition;
      ROS_INFO("Right Arm should have released the handle");
    }
    if (state == GoBacktoResPosition)
    {
      if (once == 0)
            vpDisplay::displayText(img_, vpImagePoint(15,10), "Left click to open the
fingers", vpColor::red);
      else
          vpDisplay::displayText(img_, vpImagePoint(15,10), "Left click to put back the
RArm in rest position", vpColor::red);
      if (click_done && button == vpMouseButton::button1 && once == 0) {
        romeo.getProxy()->setStiffnesses("RHand", 1.0f);
        AL::ALValue angle = 1.0;
        romeo.getProxy()->setAngles("RHand", angle, 1.);
        state = DoorOpenedAndQuit;
        click_done = false;
      }
    }
```

ANNEX 16. Pushing the door handle

```
if (state == CloseDoor)
    {
      // Open loop upward motion of the hand
      vpColVector cart_delta_pos(6, 0);
      cart_delta_pos[0] = 0.10;
      cart_delta_pos[1] = 0.03;
      double delta_t = 3;

      door_closed = true;

      static vpCartesianDisplacement moveCartesian;
      vpVelocityTwistMatrix V;
      if (moveCartesian.computeVelocity(romeo, cart_delta_pos, delta_t, "RArm", V)) {
                                           romeo.setVelocity(moveCartesian.getJointNames(),
moveCartesian.getJointVelocity());
      }
      else
      {
        romeo.stop(moveCartesian.getJointNames());
           vpDisplay::displayText(img_, vpImagePoint(15,10), "Left click to release the
handle", vpColor::red);
        if (click_done && button == vpMouseButton::button1 ) {
          state = ReleaseDoorHandle;
          click_done = false;
        }
        ROS_INFO("Right Arm should have finished to close the door");
      }
    }
```

Abstract

ROMEO is a humanoid robot aiming to become a real assistant and personal helper. In order to be able to commercialize it, numerous functions and functionalities still have to be created.

This project was composed of a short part on the creation of a simulation environment for ROMEO, before being dropped due to a compatibility problem. Then in order to develop a function to open a door with ROMEO, the creation of a door handle localization program was necessary. This report present the different steps that were necessary for the realization of a function to open a door with ROMEO.