

# Final Master Thesis

**Master's Degree in Automatic Control and Robotics**

## **Mobile manipulation with the TIAGo robot: perception and task manager**

### **Memory**

June 24, 2019

**Author:** Miller Stiven Espinosa Muñoz

**Advisor:** Prof. Dr. Jan Rosell Gratacòs

**Date:** 06/2019



Escola Tècnica Superior  
d'Enginyeria Industrial de Barcelona





## Abstract

This project aims to design and implement a solution for a complex mobile manipulation task using PAL Robotics' robot TIAGo. The goal is to use the platform as an assistive robot, making it capable of picking a chosen object from a certain location and moving it to another. More precisely the task in which the project will focus will be to pick a certain soda can from a table with several of them on it and pour its contents into a glass placed on another table for the user to have his/her drink . For that, all the perception, interaction, planning and mobile capabilities of the robotic platform will be exploited in order to develop a suitable and complete solution.

The generated solution is a modular method that can be launched as a whole process to complete the entire challenge, but this solution also gives the possibility of using each one of the modules independently. This way it is feasible to easily integrate them to other processes in order to complete other similar tasks, making the packages more versatile and adaptable.

The entire project has been divided in two parts. One focused on developing the packages in charge of navigation and arm manipulation, carried out by Xavier Garcia Peroy. The other one, focused on developing perception and task management part, and described in this report.

For the perception part, some computer vision capabilities have been implemented using TIAGo's camera. These capabilities were added in order to get knowledge about the objects with which the robot needs to interact. In this case, these objects were cans of soda which the robot needed to detect and, using some image processing steps, determine their position in order to pick the can of soda chosen by the user.

On the other hand, for task management part, a solution based on *behavior trees* was developed. This solution has been done in a modular way, removing a big part of the complexity of executing each necessary task to achieve the goal of this project, and also decreasing the necessity of going deep in programming in order to make changes and check partial functionalities. That has been achieved with the use of the *BehaviorTree* library and *Groot*, a visual tool that has allowed to create the task manager functionalities graphically.

Both solutions have been checked to work properly in order to achieve the established goal. However, specially in perception part, some functionalities should still be improved in order to increase the robustness of the method and decrease some limitations. Future work is suggested in order to make these enhancements.



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Motivation . . . . .	8
1.2	Objectives . . . . .	9
1.3	Scope of the project and methodology . . . . .	10
1.4	State of the art . . . . .	11
1.4.1	Perception . . . . .	11
1.4.2	Task manager . . . . .	12
1.5	<i>TIAGo</i> robot description . . . . .	14
<b>2</b>	<b>Perception</b>	<b>17</b>
2.1	Introduction . . . . .	17
2.2	Image processing . . . . .	17
2.2.1	Image acquisition . . . . .	17
2.2.2	Segmentation . . . . .	18
2.2.3	Feature extraction . . . . .	21
2.2.4	Image processing results . . . . .	24
2.2.5	Pseudocode . . . . .	24
2.3	Pose estimation . . . . .	24
2.3.1	Results and problems of pose estimation . . . . .	26
2.3.2	Can position and orientation optimization . . . . .	26
2.4	Implementation inside the task manager . . . . .	29
<b>3</b>	<b>Task Manager</b>	<b>31</b>
3.1	Introduction . . . . .	31
3.2	Behavior Tree Library Theory . . . . .	31
3.2.1	Introduction . . . . .	31
3.2.2	Type of nodes . . . . .	32
3.2.3	Ports . . . . .	33
3.3	Behavior Tree inside ROS . . . . .	33
3.3.1	How it works inside ROS . . . . .	33
3.3.2	How to implement behavior tree nodes . . . . .	34
3.3.3	How to build the logic process . . . . .	36
3.4	Structure used in this project . . . . .	38
3.4.1	General view . . . . .	40
3.4.2	System process explanation . . . . .	40
<b>4</b>	<b>Experiments and Results</b>	<b>49</b>
4.1	Perception experiments . . . . .	49
4.2	Task manager experiments . . . . .	54
4.3	General process experiments . . . . .	59
<b>5</b>	<b>User manual</b>	<b>61</b>
5.1	Required actions before execution the entire system process . . . . .	61
5.1.1	Environment setup . . . . .	61
5.1.2	Connection to <i>TIAGo</i> . . . . .	62

5.1.3	Map building . . . . .	62
5.2	How to reproduce the project . . . . .	63
<b>6</b>	<b>Costs</b>	<b>65</b>
<b>7</b>	<b>Environmental and social impact</b>	<b>67</b>
7.1	Environmental impact . . . . .	67
7.2	Social impact . . . . .	67
<b>8</b>	<b>Conclusions</b>	<b>69</b>
8.1	Specific conclusions . . . . .	69
8.1.1	Perception . . . . .	69
8.1.2	Task manager . . . . .	70
8.2	General conclusions . . . . .	70
<b>9</b>	<b>Future work</b>	<b>73</b>
9.1	Perception . . . . .	73
9.2	Task manager . . . . .	73
9.3	General process . . . . .	73
<b>10</b>	<b>Acknowledgements</b>	<b>75</b>
<b>References</b>		<b>76</b>
<b>A</b>	<b>Code Repository</b>	<b>79</b>
<b>B</b>	<b>XML file of the project</b>	<b>81</b>
<b>C</b>	<b>Pose estimation algorithm</b>	<b>87</b>
C.1	The feature vector . . . . .	87
C.2	Analytic solution . . . . .	89
<b>D</b>	<b>Optimization of pose estimation</b>	<b>93</b>
D.1	Pose correction . . . . .	93
D.2	Orientation correction . . . . .	94

## List of Figures

1	Image of the hackathon competition [1] at the <i>IROS 2019</i> . . . . .	8
2	Example of state diagram of a Finite State Machine (FSM). . . . .	12
3	<i>TIAGo</i> robot. . . . .	15
4	Image processing steps . . . . .	17
5	Original images. . . . .	18
6	Gray images. . . . .	19
7	Binarized images. . . . .	19
8	Can parts in image processing. . . . .	20
9	Contours of the top of the can. . . . .	20
10	Necessary points to perform pose estimation algorithm. . . . .	22
11	Steps to obtain $S_i$ points. . . . .	23

12	Detection of each can and addition of their labels. . . . .	25
13	Soda cans labeled with its number. This number is shown to the user to let him/her choose the drink. . . . .	25
14	Taken capture. . . . .	27
15	Visual result without applying any correction process. . . . .	28
16	Visual result without applying any correction process, from TIAGo's view perspective. . . . .	28
17	Result after applying position and orientation correction. . . . .	28
18	Result after applying position and orientation correction, from TIAGo's view perspective. . . . .	28
19	Result applying just orientation's correction, from TIAGo's view perspective. . . . .	28
20	Result applying just orientation's correction. . . . .	28
21	Result after applying both optimization processes. . . . .	29
22	Result after applying both optimization processes, from TIAGo's view perspective. . . . .	29
23	Blank Groot interface. . . . .	37
24	Blank Groot interface. . . . .	38
25	System Behavior Tree created in Groot GUI. . . . .	39
26	Project's behavior tree general view with sub-trees. . . . .	41
27	Open bar. . . . .	44
28	Serving table user interaction. . . . .	44
29	Go to sensing table. . . . .	45
30	Ask user to choose drink. . . . .	45
31	Grasp can. . . . .	46
32	Go to serving table. . . . .	46
33	Serving drink. . . . .	47
34	Experiment to check orientation with sample 1. A ROS topic was used to publish a message of type visualization_msgs/MarkerArray. . . . .	50
35	Experiment to check orientation with sample 2. A ROS topic was used to publish a message of type visualization_msgs/MarkerArray. . . . .	51
36	Experiment to check orientation with sample 1. A ROS topic was used to publish a message of type visualization_msgs/MarkerArray. . . . .	52
37	Experiment to check orientation with sample 2. A ROS topic was used to publish a message of type visualization_msgs/MarkerArray. . . . .	53
38	Experiment with sample 1. It shows if moving the grasping tool of TIAGo, it is possible to make the can fit between grasping fingers with the position of each can given by perception module. . . . .	55
39	Experiment with sample 2. It shows if moving the grasping tool of TIAGo, it is possible to make the can fit between grasping fingers with the position of each can given by perception module. . . . .	56
40	Output printed in terminal for each experiment. . . . .	57
41	Behavior tree used in <i>goToSensingTable</i> sub-tree modified in order to force a failure. . . . .	58
42	Observation of a point <i>P</i> . . . . .	88
43	Problem setup. . . . .	89

**List of Tables**

1	These errors are computed for sample 1. They are obtained by comparing the position of the soda cans obtained from perception module with the position obtained from aruco detector. . . . .	52
2	These errors are computed for sample 2. They are obtained by comparing the position of the soda cans obtained from perception module with the position obtained from aruco detector. . . . .	53
3	Calculation of the final cost of the project. (Variable costs of <i>TIAGo</i> and lab computers have been computed dividing their fixed cost by their life expectancy in hours. Variable costs of electric consumption of each systems have been computed multiplying the power consumption of each machine by the average price of electricity described before). . . . .	66

## 1 Introduction

Robotics is one of the engineering areas that has been growing more and more in the past years. Everyday the news and social media have new articles presenting new amazing systems that can perform more complicated tasks. In the past, robots were almost only present in industry focusing in working on big repetitive manufacturing chains and in the military world. They were strictly separated from workers and doomed to do the same operations over and over. Lately, it has been more common to see robots away from industrial environments, like in hospitals, hotels, in the sky or even in our homes.

For this reason *Service Robotics* is one of the fields in robot engineering that has been gaining a more important role last years. With the appearance of home robots that conquered successfully the market and became very well spread among society like *Roomba*, companies all over the globe have started to invest more money and efforts into generating new robotics products. In the last decade, the number of investigations around it and therefore the number of applications has increased year after year making it a main topic in laboratories and in the lead R&D research companies of the field.

Service robots, as described by the International Organization for Standardization, are those robotic solutions “that perform useful tasks for humans or equipment excluding industrial automation applications” [2]. For that reason, this kind of robots must be capable of performing tasks based on the current state of its environment, being capable of sensing it and deciding autonomously. To achieve such an autonomous behaviour this kind of robots must combine theory and techniques coming from a wide range of different fields in order to perform a certain task which might be perceived as quite simple by a human being. Knowledge and techniques from fields like computer vision, machine learning, artificial intelligence, task planning and many others must be combined successfully to solve correctly the task. All of them must be studied and exploited in order to provide the robotic platform with all the tools necessary to complete its assignment.

This project will focus on completing a service task, in particular its goal will be to use a mobile manipulator to serve a drink to a client. The robot selected is *TIAGo*, the collaborative mobile manipulator developed by the Barcelona company *PAL Robotics*. According to the description of the robot in the company’s web page the *TIAGo* robot “makes it immensely easy for developers to create their own applications and is open to customization and expansion. Fully ROS-enabled<sup>1</sup>, it integrates manipulation, perception, navigation and HRI skills to suit multiple scenarios”[4]. Being a fully ROS enabled robot and all the different features that it integrates out of the box make this robot a very good option to try to complete a task of such characteristics

Nowadays there are several examples of robots designed specifically to complete the task of serving food and drinks to customers in bars and restaurants. Videos [5] and [6] depict a couple of examples of these kind of robots. For now these robots do not have many cognitive capabilities as they are mainly “transporting” robots which carry the orders of

---

<sup>1</sup>ROS: Robotics Operating System.[3]

the customers in a tray from the kitchen to the table. Therefore their main capabilities are focuses on navigation through crowded environments. They focus on being robust avoiding collisions and reaching the final goal correctly but they do not exploit other functions like moving objects using any kind of actuator or perception abilities to recognize the order they have to offer to each customer. Their interaction with the users is very reduced.

In a broad way the goal of this project is the same as the ones presented previously but the steps are a little more complicated as the main idea is not only to transport the orders of the client from the kitchen to the table but to let *TIAGo* have more decision power. The process consists in letting the user select a can from a bunch of them placed on top of a table by making *TIAGo* detect and identify them using its perception capabilities. When the user has chosen its drink, the robot must look for the best approach to pick it and execute the most suitable movement to grasp the can. Afterwards the robot must navigate towards a serving table where an empty plastic cup has been placed. Then it should pour the contents of the can into the cup for the user to enjoy his or her drink. This process must be repeated every time a customer asks for a new drink.

## 1.1 Motivation

The main motivation behind this project came by having the opportunity of participating in a challenge on the *2018 IEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, as a member of the robotics lab team of the *Institute of Industrial and Control Engineering (IOC-UPC)*. The challenge proposed was to use the *PAL Robotics* robot, *TIAGo*, to manipulate and serve cans of soda as demanded by users, in a way, implementing a "waiter" robot [7].



Figure 1: Image of the hackathon competition [1] at the *IROS 2019*.

For the completion of this project, a team of doctoral and master studies students developed different solutions together to tackle all the tasks involved in the process and worked against the clock during a couple of months to implement a suitable solution for the challenge. As part of this team I had a chance of following the development of the project closely, learning about all the different parts implemented and being responsible of some of them. Due to the time lack and other technical problems faced during the hackathon it was not possible to complete a fully operational solution for the proposed day and the project stood half completed after the challenge passed.

Then, after some months, my partner Xavier Garcia and I got a proposition from Jan Rosell, the professor leading this project, to couple up and work together in order to finish all the different sides of the project that were left to do. The motivation of this final thesis work was to complete, document and put together all the necessary blocks to finish the project.

The aim of the project was to test and prove that a robotic manipulator platform, specifically the *TIAGo* robot, could be used to complete such a mobile manipulation task. To test that, using already developed packages and new solutions, the hackathon challenge could be completed and a solution could be found to make the robot serve drinks to users as proposed. The tools generated could be afterwards used in other projects and tasks as packages to interface *TIAGo*. For *IOC* it would be also very interesting to use this project as a demo presentation of the mobile manipulation capabilities developed by students in the laboratory of the institute.

## 1.2 Objectives

Specifically, the project aims to design and implement a solution for a complex mobile manipulation task using *PAL Robotics*' robot *TIAGo*. The goal is to use *TIAGo* as an assistive robot, making it capable of picking a desired object from a certain location and moving it to another. More precisely the task in which the project will focus will be to pick a certain soda can from a table with several of them on it and serve its contents into a glass for the user to have his/her drink. For that, all the perception, task manager, planning and mobile capabilities of the robotic platform will be exploited in order to develop a suitable and complete solution. The idea is to generate a modular method that can be launched as a whole process to complete the entire challenge but also that each one of the modules can be used independently. This would make it possible to easily integrate them to other processes in order to complete other similar tasks, making the packages more versatile and adaptable.

This particular part of the project will be focused on developing some perception capabilities to detect an specific object, in this case a soda can, that is wanted to be located and grasped. In order to do that, it will be essential to implement an image processing algorithm that detects some key points of each object and then, apply another algorithm to estimate the real position of the objects. On the other hand, this part will be also centered on the application of a task manager with a useful tool called *Groot*. The aim of task manager module is to plan the sequence of actions that the robot has to perform in order to achieve its goal: to pick a can of soda and serve it to the user in a glass.

Aside from this main technical goals, another objective is to generate a clear and understandable code that can be used as bases for other projects whose aim is to enhance the current solution or work in different problems with similar tools. It will also be demanded to have a suitable documentation of the code and the methods be implemented to be as easily reusable as possible.

### 1.3 Scope of the project and methodology

The scope of this project is to combine and correctly use different tools provided by the robot manufacturer or other open source third parties in order to successfully complete the task proposed. It is not within its scope to develop new innovative techniques and solutions but to combine and adapt already existing tools to fulfill the objective established.

The methodology followed to face this project was to first structure it into 4 blocks. They are introduced next.

- First one is in charge of navigation and path planning of the mobile base. It will gather all the implementation necessary to successfully map the environment and navigate it to complete the serving process.
- Second block focuses into the manipulation of the cans using the 7 DOF robotic arm. This will be in charge of performing the picking and serving motions involved in the process.
- Third one consists in exploiting the perception capabilities of the platform. Using the RGB information of the camera placed in the head of *TIAGo* the robot will have to detect the position of the cans and other elements in the scenario.
- Finally the last block is the manager of all the rest of the task. This block will cover the planning of the process and the coordination between all the task in order to serve the drinks successfully. It will also include a human robot interface to make it easier for the users to communicate with and understand *TIAGo* along the process.

Each one of the two student involved in the project will focus in two of these four sections. As mentioned, this project will focus on the last two blocks concerning perception and the task manager and this report will be covering and explaining them.

It is important to remark that the third point was already part of the hackathon project and was developed under my lead. The solution implemented for the competition day did not work quite well due to the brightness of the scene, the low robustness of the detection method and the low accuracy of the pose estimation of the cans. In this project some enhancement have done in order to improve the robustness of detection and the readability of the code, but following the same guideline from the hackathon.

On the other hand, point four, regarding the task manager, has been completely redesigned and developed for this current project to make it modular and reusable. A better solution was found, which avoids a spaghetti code solution and improve the readability and error checking of the implementation.

Once structured the workload, the methodology planned and the work plan followed was the one presented next.

1. Firstly, the solutions and packages developed for the hackathon had to be analyzed and assessed. The rest of tools provided by the manufacturer as well as other open source solutions that could be useful in the project were explored and tested.
2. Next step was to decide how each one of the blocks described before was going to be faced and which would be the frameworks and tools used to work on each one of them.
3. Then, each team member had to work on completing its assigned parts. The results on each block for this first implementation step were tested on simulation to prove its performance and functionality.
4. After that, all the packages developed had to be merged again into one single project that would contain the final solution. The main goal of this step was that all the different packages had to end up correctly integrated in a way that the task manager could interface and use them and they could communicate all the necessary information between them.
5. The final step followed was to test the solution in the real robot with a real scenario in the *IOC* robotics lab.

## 1.4 State of the art

In this part of the report a small asses on the state of the art for Perception and Task Manager will be presented.

### 1.4.1 Perception

Nowadays, there are many algorithms that allow to detect and estimate the position of an object. The trend now is to use methods based in Convolutional Neural Networks for this purpose, but in this project it is wanted something simple and robust, something that can be used in more structured scenarios. That is why, classical methods have been chosen.

OpenCV [8] is an open source library that has different programming functions related to computer vision. It also has multiple solutions that can be used to solve computer vision challenges. One of them is the *Real Time pose estimation of a textured object* [9], which finds the position of a textured object extracting ORB features and descriptors from the scene and then it matches the scene descriptors with the descriptors of a model. This is performed using Flann matched. Finally, it estimates the position using *PnP + Ransac*. Additionally a Linear Kalman Filter is applied for bad poses rejection. The disadvantage of this solution is that it just work with textured objects and a 3D model of the object is needed.

There exist other multiple solutions that can be tested in this problem, but a *Pose Estimation* algorithm from *Modelling, Planning and Control* book [10] has been applied because it was considered a good option to estimate the position of a can of soda, because with this solution

it is not necessary to have a 3D model from the object that is wanted to be located, just its dimensions, and it was not needed to use many features in order to perform the image processing. To sum up, this algorithm has been chosen due to its simplicity in the image processing part.

### 1.4.2 Task manager

Every time we want to carry out a project that requires to program a system process, as its complexity increases, the implemented programming code becomes more and more complicated, it becomes more difficult to debug and it is increasingly difficult to scale. There are mathematical models of computation that help to solve these kind of problems. One of them is the Finite State Machine (FSM), which starts in an *initial state* and transitions to other states depending on what state it is at each moment and what input it is given.

This finite state machine can be defined in a state diagram like the one shown in figure 2, having an overview of the system's operation. In addition, there are some libraries that allow from a FSM generate the system behavior. Examples of this type of library are:

- Tinyfsm [11]: library written in c++ that allows to program the behavior of a system from a state machine.
- Transitions [12]: library in python that also allows to program the behavior of the system from a state machine. But unlike the library *tinyfsm*, this library can define the operations defined in a state diagram using a JSON file, making the task of creating the system's behavior much simpler and easier to scale. In listing 1 the transformation from the finite state machine shown in figure 2 to a JSON file can be observed.

With these programming tool it is just necessary to think about the behavior of the system and the programming of each of the functional parts without having to be worried about an increasingly complex and spaghetti programming code, having everything much more clear and simple.

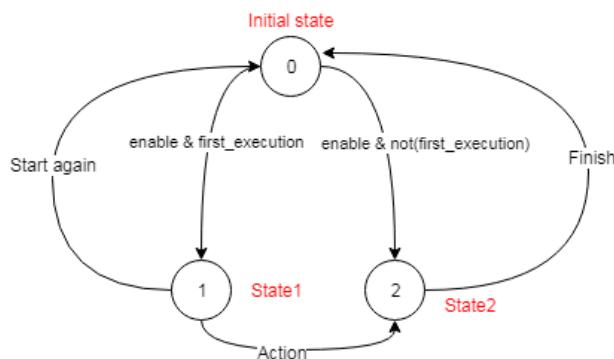


Figure 2: Example of state diagram of a Finite State Machine (FSM).

Listing 1: JSON file equivalent to finite state machine previously defined in figure 2.

```

1 {
2   "model-class-name": "TestFsm",
3   "states": ["initial_state", "state_1", "state_2"],
4   "initial-state": "initial_state",
5   "transitions": [
6     {"trigger": "enable", "source": "initial_state", "dest": "state_1", "conditions": ["is_first_execution"], "after": ["action_function_1"]},
7     {"trigger": "enable", "source": "initial_state", "dest": "state_2", "conditions": ["is_not_first_execution"], "after": ["action_function_2"]},
8     {"trigger": "start_again", "source": "state_1", "dest": "initial_state", "conditions": [], "after": ["action_function_3"]},
9     {"trigger": "action", "source": "state_1", "dest": "state_2", "conditions": [], "after": ["action_function_4"]},
10    {"trigger": "finish", "source": "state_2", "dest": "initial_state", "conditions": [], "after": ["action_function_5"]}
11  ]
12 }

```

On the other hand, another similar but more powerful tool is behavior tree, very used in robotics. Behavior tree is a way to structure the process of different tasks in a modular way. This characteristic is very important when the entire process starts to become more complex.

The main advantages of using a behavior tree are the simplicity, division in little tasks, easy scalability, the ability to create complex systems, and the easy understanding both to create new systems and understand created ones.

As finite state machine (FSM), behavior tree would be another option better than hard coding to create a modular system to execute a task manager. But behavior tree model is superior than FSM due to the following reasons:

- Behavior Tree (BT) allows to isolate better the modules. Unlike FSM, BT has not transitions, so it does not need a source node fulfill a condition to continue with the process. BT uses sequences and they can return RUNNING, SUCCESS or FAILURE states, so it is not necessary to define every transition to go to the following node.
- BT nodes can communicate among them using ports.
- It is easier to debug with BT, because apart from its better isolating capabilities, it exists decorator nodes, that allows to manipulate the behavior tree easily to send a specific response to a group of BT nodes, debug their functionalities and to make more expressive the behavior tree.

- The more complex the system becomes, the more probable is for FSM to create a spaghetti state machine, and this increases when tasks are divided in smaller ones.
- If it is necessary, a BT can run two different nodes at once. In order to do this with FSM, it is necessary to create another different FSM, being less readable and understandable.
- Finally, a good advantage of BT over FSM is the existence of visual editors that transform a graphical Behavior Tree into an XML description file, understandable by the BT library used to describe the BT structure.

Finally, Behavior Tree was chosen as the best tool to implement the task manager of this project.

In order to implement a behavior tree, an open-source library, called *BehaviorTree.CPP* [13]. On the other hand, for the behavior tree description a GUI editor, called Groot [14], was used.

## 1.5 TIAGo robot description

A little description of the robot will be provided in order to have a better picture of the different characteristics and features of the robot.

From the hardware point of view the robot incorporates several different elements of interest for this project. They are described next from top to bottom. The first element is a mobile head with a RGB-D camera inside it. The head can be moved using two motors on the its base to perform tilt and pan rotations. These movements can be used to focus the camera view range to different regions without moving the robot. This increases the visual range of the perception area of the camera. Just below the head, on the base of the neck, the robot incorporates as a stereo microphone for interaction and a speaker that can be used to reproduce different sounds.

Going down to the torso of the robot, the most important element is the 7 degrees of freedom arm mounted on its chest. The first joint can be used to rotate the shoulder over the ground plane as an inverted 7DOF industrial arm. In particular, this robot has changed its mechanical limits to present a left-handed configuration instead of the standard right-handed one. The end-effector base incorporates a force/torque sensor to check the pay load on the end of the arm. The final tool mounted to the arm is a two claw gripper as the one in Figure [3].

Following from top to bottom, a prismatic joints is found on top of the mobile base. This link can be extended and retracted to change the height of the robot by lifting or lowering the torso. Finally, below this link, the mobile base is mounted. This is a differential drive mobile base that gathers all the necessary elements for navigation. It incorporates a belt of LEDs that can be managed to perform different lights and patterns, a laser range-finder sensor and two rear sonar sensors to successfully navigate the base.



Figure 3: *TIAGo* robot.

From the software point of view, *TIAGo* uses as operating system the 64-bits version of *Ubuntu*, *RT Preempt real-time framework*. The robot uses *ROS* as a robotics middleware between the software layer and the *Ubuntu* OS. A robotics middleware is a communication layer between the software solutions developed by the user and the different drives of the OS used to control the robot. In particular *ROS* is an open source layer that provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management.

Over this *ROS* layer, the robot includes a lot of different tools and features that are very useful to developers. There are a lot of different packages already installed in the robot that can be helpful for so many different tasks from controlling the base to recognizing human faces using the camera images. During the report the more relevant for this project will be described and mentioned.



## 2 Perception

### 2.1 Introduction

The main objective of this module is to estimate the position of the cans of soda to be grasped. In order to do that, some *image processing* methods are applied to detect the cans and to obtain all the necessary information required for the *pose estimation* algorithm to obtain the position of each of the cans. Then, these positions are sent to task manager module, which call the arm's module and gives to it the position of the object that must be grasped.

This module is implemented as a ROS node and called by the task manager module with the use of a ROS service.

### 2.2 Image processing

In order to estimate the position of each of the targeted objects, it is necessary to apply some image processing steps to obtain some key points in the image plane, as a part of the input data of the algorithm in charge of estimating the position of the cans of soda. These image processing steps are summarized in Figure 4.

#### 2.2.1 Image acquisition

First of all, it is necessary to capture the image showing all the cans that can be selected by the user.

ROS gives the possibility of subscribing/publishing information through topics, relevant to the robot and the devices being used.

For instance, there are topics related to the information from the camera that TIAGo has. For instance, the topic `/xtion/rgb/image_rect_color` is the topic with the rectified color image. When the perception module is called, it is subscribed to this topic until it gets an image from the camera. This image is received in a message of type `sensor_msgs/Image`, that is later converted into a *Mat* variable in order to perform all the image processing steps.

As an example, Figure 5 shows two images taken by TIAGo's camera and published at topic `/xtion/rgb/image_rect_color`. From now on, these images are going to be used in order to show each of the results obtained at each image processing step.

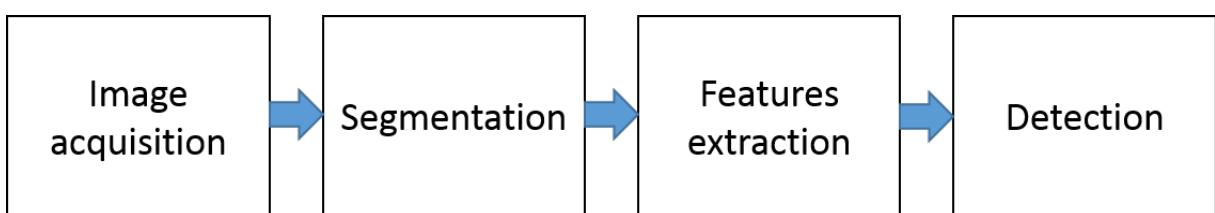


Figure 4: Image processing steps



Figure 5: Original images.

### 2.2.2 Segmentation

When the image is acquired, the segmentation process starts. This process executes the following steps.

#### From color to gray

The original image is an image with color. So, to execute the following segmentation process it is necessary to transform image from color to gray. The result of this transformation is shown in Figure 6.

#### Binarization

In order to isolate the cans from the rest of the scene, different alternative were attempted, being the binarization the best way to do it. Two results of a binarization are shown in Figure 7.

The main reason why binarization works well is because the high gray intensity that the top of the cans has in relation to the rest of the objects scene. So, this way it is easier to isolate the cans from the scene. Nevertheless, this is not enough to detect if an object is a can or not. It is also necessary to extract other features of the can, as the location of the hole on its top (cans are assumed to be opened, ready to be served).

It is possible to observe in Figure 7a that the binarization also isolates some bright regions from the scene, possibly detected as cans. For this reason, it is necessary to add an image processing procedure to have a better can isolation.

#### Contours

After binarization it is possible, with some of the *Open CV* functions, to fit an ellipse to the contours obtained.

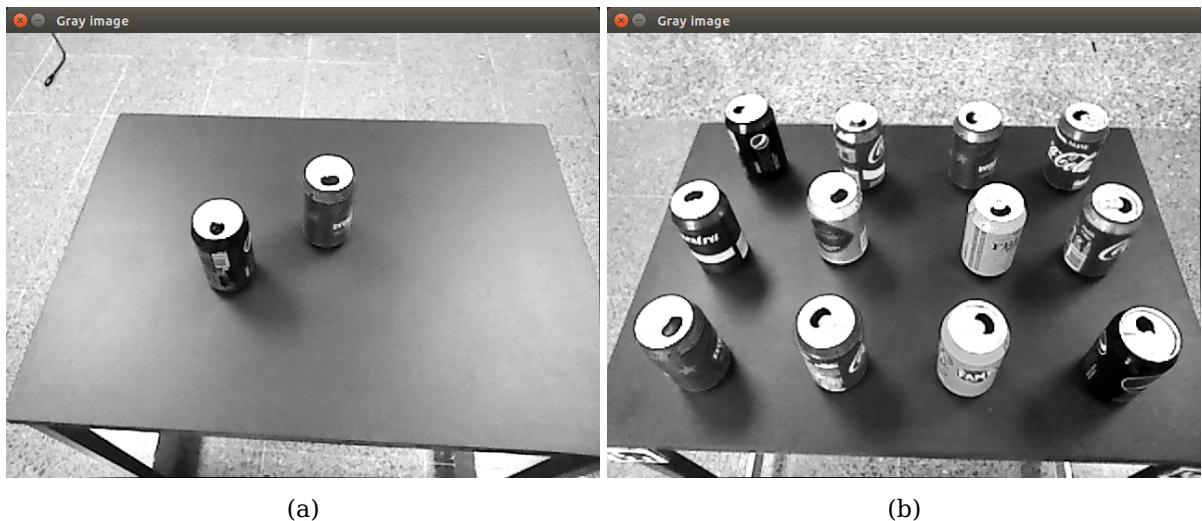


Figure 6: Gray images.

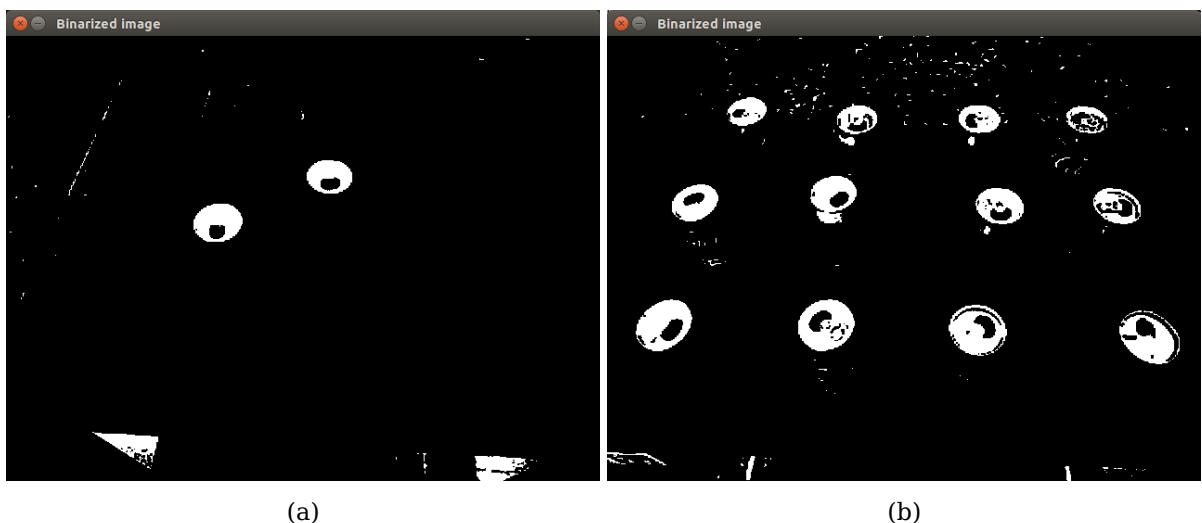
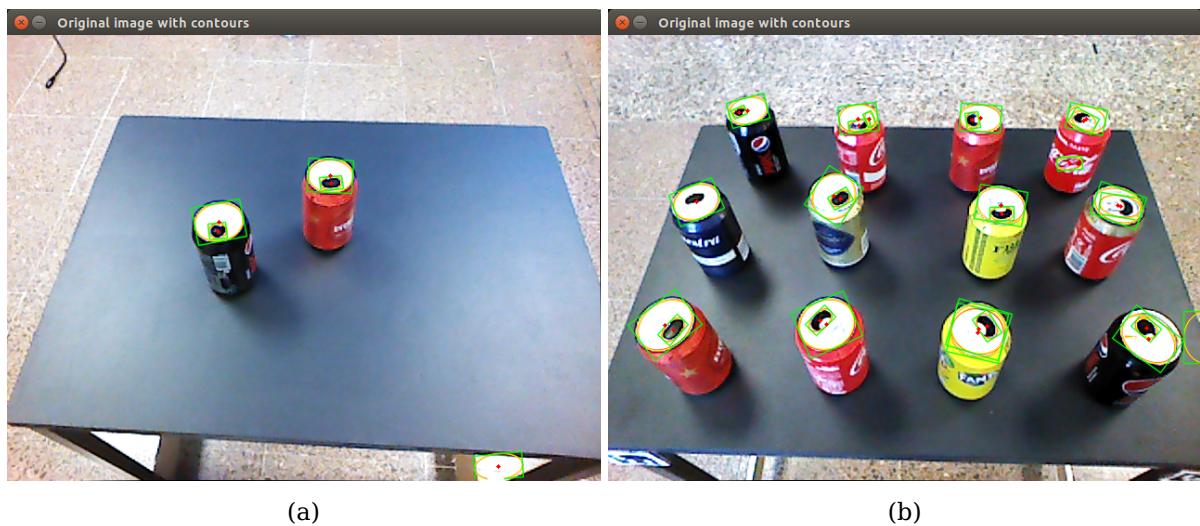


Figure 7: Binarized images.



Figure 8: Can parts in image processing.



(a)

(b)

Figure 9: Contours of the top of the can.

### 2.2.3 Feature extraction

As binarization it is not enough to isolate and detect cans, it is necessary to extract some features that cans have, like the existence of a hole at its top. Fig. 8 shows the top contour and the hole of a coke can.

#### Distinction between can's top and hole

Figures 9a and 9b show the contours detected for both can's tops and holes (and some false detection too).

In order to know if some detection is a can or not, it must have a hole inside its top, i.e. it should have a can's hole contour inside a can's top contour. So, first of all it is necessary to distinguish between a top and a hole. To do that, each contour will be compared with the rest of contours.

If the contour that is being evaluated at the moment (e.g. contour A), is bigger, and the center of the other contour that is being evaluated (e.g. contour B) is in inside contour A, contour A will be assigned as *the top* of the can and contour B as *the hole* of that can. But if another possible hole is detected and it has more possibilities of being a hole, because it has higher black pixels rate than the previous assigned hole, it will substitute the previous hole. On the other hand, in order to improve the robustness of a can detection, a second round to detect holes inside contours was added, segmenting each top of can and looking for a black hole by searching clusters of black pixels.

#### Feature vector points at image plane

Now, there is information about the top and hole contours and their center. With that, it is possible to find the four coplanar points ( $S_1$  to  $S_4$ ) required to estimate the position of each can. These points correspond to the intersection with the contour of the top of the can of the  $x$  and  $y$  axis of a reference frame located on the plane of the top side and centered at its middle point (Fig. 10)

#### Obtaining points $S_1$ and $S_3$

To extract that points it is necessary to join the center of the top of the can  $\mathbf{X}$  with the center of the can's hole  $\mathbf{H}$  and create a line that intersects with the ellipse of the top of the can. This point is called  $S_1$ , then  $S_3$  is created by drawing a line in the opposite direction and intersecting with the ellipse of the top of the can again. It is possible to observe this behavior in the transition from figure 11a to figure 11b, and it is described in expression 1.

$$(s_1, s_3) = \text{intersection}(\text{line}(X, H), \text{ellipse}) \quad (1)$$

#### Compute vectors $v_i$ and $u$

Now, in order to obtain the other two points  $S_2$  and  $S_4$ , it is necessary to perform some computations. First of all, it is necessary to compute local vectors  $v_i$  and  $u$ .

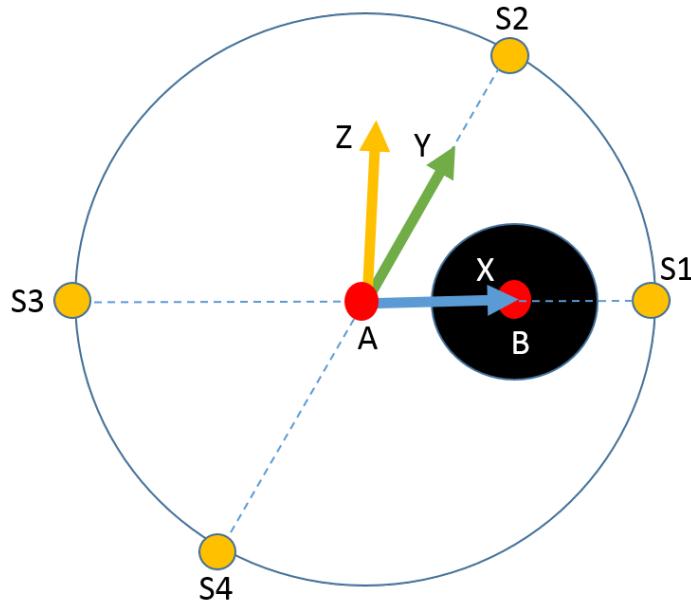


Figure 10: Necessary points to perform pose estimation algorithm.

To obtain  $v_i$  and  $u$  it is required to take into account points  $P=\{P_1, P_2, P_3, P_4\}$  and point  $X$ . These points come from the bounding box that delimits each of the can's top:

- $P_i$  are the bounding box vertices.
- $X$  is the bounding box center.

Then:

- $u$  is the vector created by joining  $X$  and  $S_1$ :

$$u = S_1 - \underline{\underline{X}} \quad (2)$$

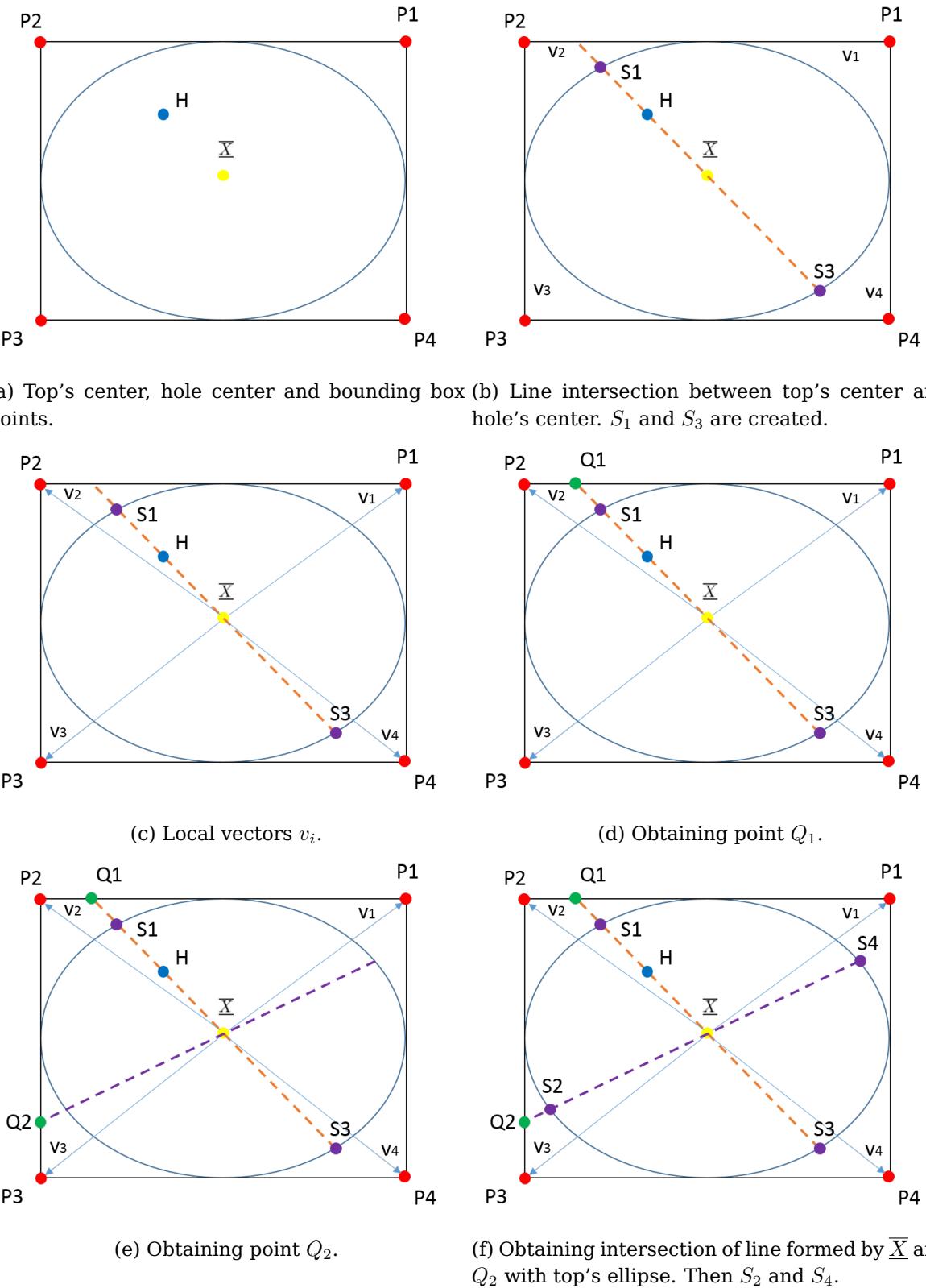
- $v_i$  is the vector joining  $X$  and  $P_i$  (Fig. 11c):

$$v_i = P_i - \underline{\underline{X}} \quad (3)$$

- $Wz_i$ : are the  $Z_i$  values of the cross products between  $v$  and  $u$ :

$$Wz_i = (v_i \times u) \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T \quad (4)$$

The sign of  $Wz_i$  is used to determine if  $S_1$  is in between  $P_i$  and  $P_{i+1}$  being  $i=\{1, 2, 3, 4\}$ : When  $Wz_j > 0$  and  $Wz_{j+1} < 0$  then  $S_1$  is in between  $P_i$  and  $P_{i+1}$ .

Figure 11: Steps to obtain  $S_i$  points.

### Computing points $Q_1$ and $Q_2$

$Q_1$  and  $Q_2$  are the points of the bounding box resulting from the intersection with perpendicular lines on the can top centered at  $X$

Point  $Q_1$  is computed as (Fig 11d):

$$Q_1 = \text{intersection}(\text{line}(P_j, P_{j+1}), \text{line}(S_1, \bar{X})) \quad (5)$$

Then,  $Q_2$  is computed taking into account the following proportional relations (Fig. 11e):

$$\begin{cases} (P_{j+1} - P_j) = \lambda(Q_1 - P_j) \\ (P_{j+2} - P_{j+1}) = \lambda(Q_2 - P_{j+1}) \end{cases} \quad (6)$$

From the first expression:

$$\lambda = \frac{\|P_{j+1} - P_j\|^2}{\langle (P_{j+1} - P_j), (Q_1 - P_j) \rangle}$$

Then:

$$Q_2 = P_{j+1} + \frac{(P_{j+2} - P_{j+1})}{\lambda}$$

### Obtaining points $S_2$ and $S_4$

Finally,  $S_2$  and  $S_4$  points are computed using  $Q_2$  (Fig. 11f):

$$(S_2, S_4) = \text{intersection}(\text{line}(Q_2, \bar{X}), \text{ellipse}) \quad (7)$$

#### 2.2.4 Image processing results

Figure 12 shows some results of this points detection with labels that indicate the can number. An image is published in a topic called *tiago\_sensing/image* to show the cans and its labels to the user, to let him choose which drink he/she wants (Fig. 13).

#### 2.2.5 Pseudocode

The pseudocode that follows (Algorithm 1) shows the procedure used to implement this part of the perception module.

### 2.3 Pose estimation

Once the image processing necessary to detect the four coplanar points on the top of the can is done, they are used to compute the position of each of the cans in the scene. The development of the method used can be found in Appendix C.

**Algorithm 1** Image processing

---

```

1: procedure ImageProcessing
2:   image  $\leftarrow$  getImage()
3:   grayImage  $\leftarrow$  fromColorToGray(image)
4:   binarizedImage  $\leftarrow$  binarization(grayImage)
5:   contours  $\leftarrow$  findContours(binarizedImage)
6:   elementsDetected  $\leftarrow$  detectElements(contours, image)
7:   [cansDetected, holesDetected]  $\leftarrow$  distinguishCanAndHole
8:   [S]  $\leftarrow$  findPoints([cansDetected, holesDetected])

```

---

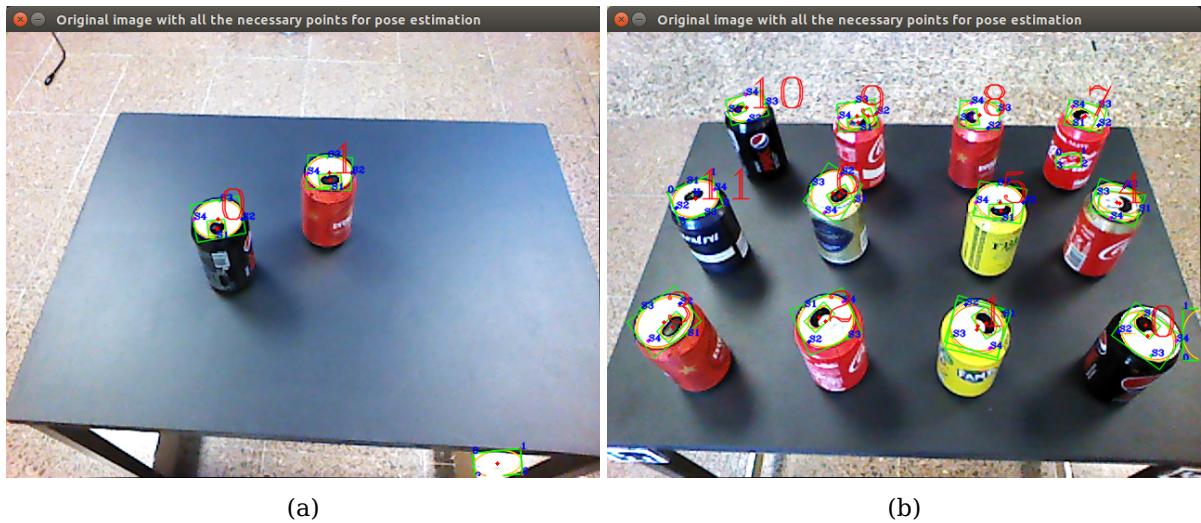


Figure 12: Detection of each can and addition of their labels.



Figure 13: Soda cans labeled with its number. This number is shown to the user to let him/her choose the drink.

The algorithm implementing this method requires three inputs:

- **The camera parameters**

The intrinsic parameters of the camera (the focal length and the principal point are required). These data are extracted according to the message type [sensor\\_msgs/CameraInfo.msg](#), from matrix P, that is the intrinsic matrix of the camera.

- **The position of each point  $S_i$  in pixels.**

These key points  $S_i$  have been found during the image processing part.

- **The nominal positions of the points  $S_i$  w.r.t the origin of the can.**

It is also mandatory to provide the nominal positions of the points  $S_i$  (shown in figure 10) in the object coordinates frame (also indicated in the figure). They are the following:

$$r_{S_1}^o = \left[ \frac{d}{2}, 0, 0 \right] r_{S_2}^o = \left[ 0, \frac{d}{2}, 0 \right] r_{S_3}^o = \left[ -\frac{d}{2}, 0, 0 \right] r_{S_4}^o = \left[ 0, -\frac{d}{2}, 0 \right], \quad (8)$$

where:

- d is the diameter of the top of the can. In this case its value is 0.026 m.

Other alternative to estimate position of soda cans was taken into account. This alternative was about using the depth camera of TIAGo, but due to the lack of information of the point cloud given by the camera, this option was discarded. So, as the information obtained was from a RGB camera, the pose estimation algorithm used was a very good choice to estimate the positions of the cans of soda that were wanted to be grasped.

### 2.3.1 Results and problems of pose estimation

Once pose estimation is implemented it was possible to obtain the position of each of the cans of the scenario. But the positions obtained from the pose estimation algorithm were not correct enough. In Figures 15 and 16 it can be observed that the position and orientation of the can are wrong.

In order to check if the cans have the correct position and orientation, a ROS topic with a message of type [visualization\\_msgs/MarkerArray.msg](#) was published with the position, orientation and dimensions of each can in a topic . This topic is used by rviz to graphically show the cans position and orientation.

### 2.3.2 Can position and orientation optimization

As the results obtained with pose estimation algorithm were not quite good, it was needed to perform two optimizations. These optimizations were performed by using a library called NLOpt [15][16].

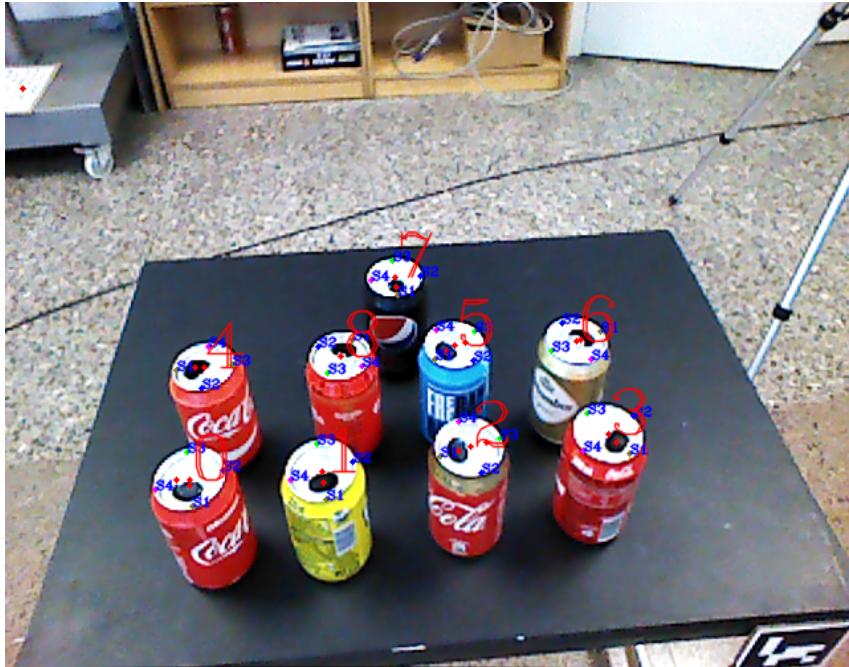


Figure 14: Taken capture.

- **Pose correction:** In order to correct the position and orientation of the cans, it was added a value to x, y and yaw components of cans position w.r.t. world. Then, the position of the cans w.r.t. camera were recomputed in order to minimize the objective function. In Appendix D there is more information about this optimization process.

In Figures 17 and 18 can be observed the results obtained after applying position and orientation correction. It can be seen that position and orientation of the cans have been corrected.

- **Correction of orientation:**

In order to correct the orientation of each can it is necessary to compute the vector from the difference between center and  $S_1$  point of each can in image coordinates, and the vector from the difference between the real position of the center of the can and  $S_1$  point projected in image coordinates. Then, the angle between these two vectors is computed. This angle should be minimized. As in the previous point, there is more information about this optimization process in Appendix D.

In Figures 19 and 20 it is shown how only the orientation was corrected having a wrong position. Moreover, it can be observed that the effect of this optimization can not be appreciated, but can be used as a reinforcement to the orientation correction of the cans.

After applying these two optimization processes the pose estimation improved a lot (the results are shown in Figures 21 and 22), but its accuracy is not good enough to obtain a robust pose estimation from any position on the drinks table. This fact is explained with more details in [conclusions](#).

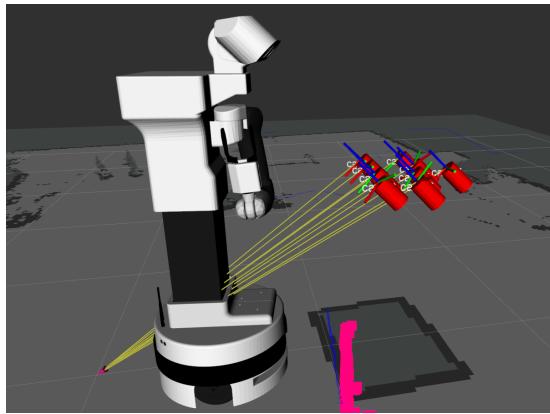


Figure 15: Visual result without applying any correction process.

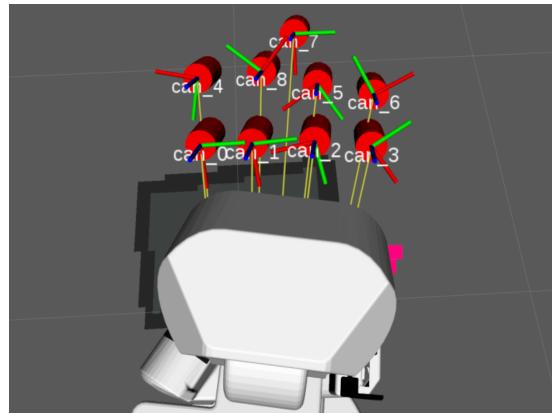


Figure 16: Visual result without applying any correction process, from TIAGo's view perspective.

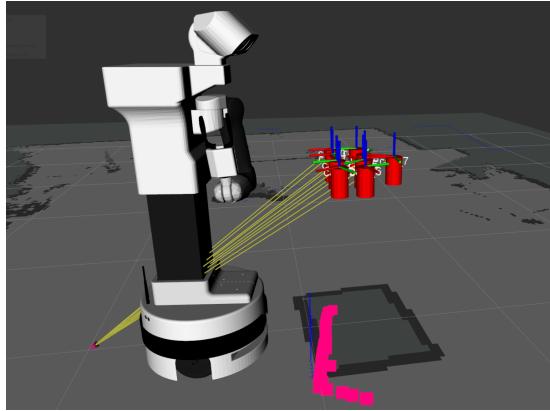


Figure 17: Result after applying position and orientation correction.

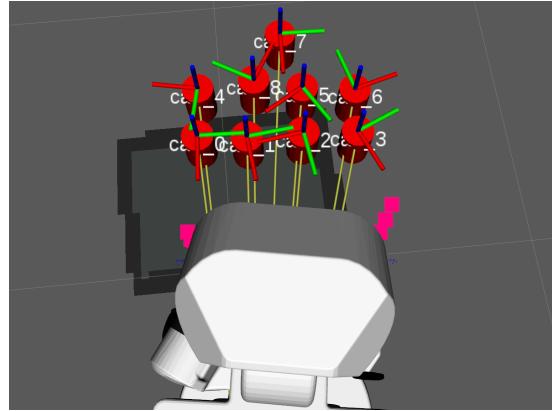


Figure 18: Result after applying position and orientation correction, from TIAGo's view perspective.

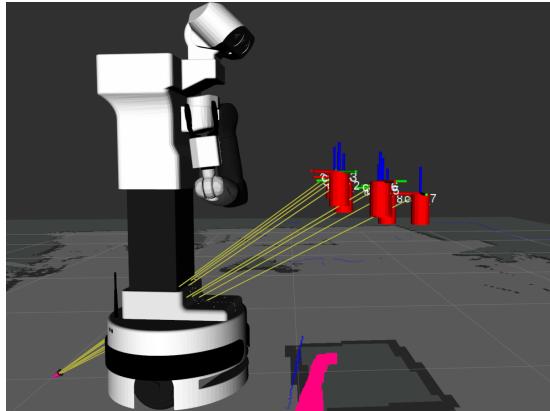


Figure 19: Result applying just orientation's correction, from TIAGo's view perspective.

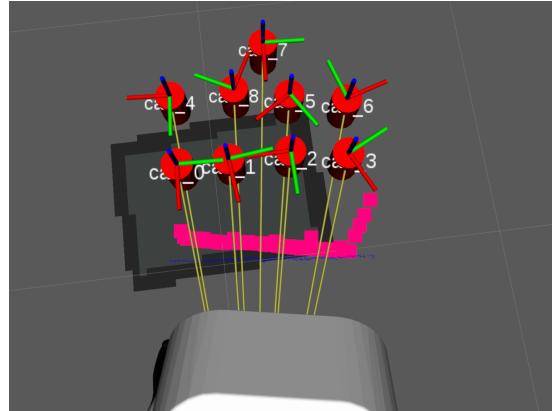


Figure 20: Result applying just orientation's correction.

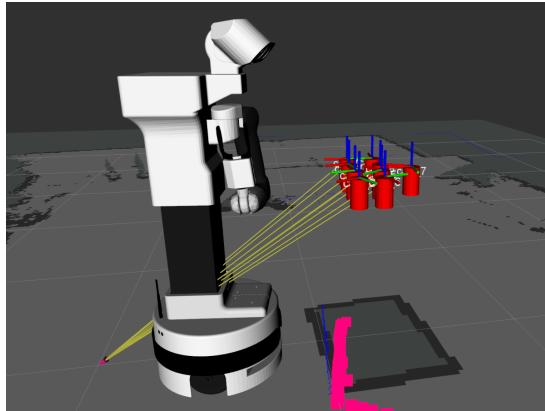


Figure 21: Result after applying both optimization processes.

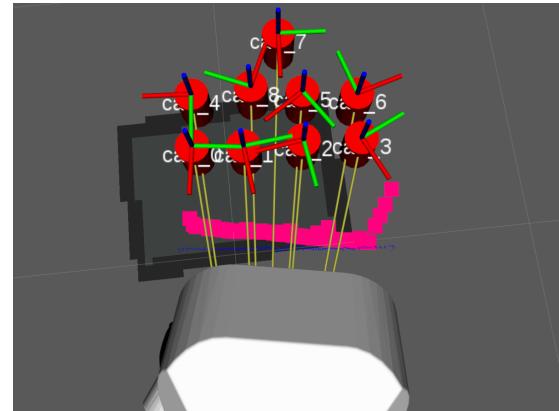


Figure 22: Result after applying both optimization processes, from TIAGo's view perspective.

## 2.4 Implementation inside the task manager

Once the perception module started to work, it was necessary to insert it to the entire system. This module has been implemented as a ROS node. This node has a service that not receives anything and returns a list of poses that corresponds with each of the cans of the scene. The poses come in a message of type [geometry\\_msgs/Pose](#).

Then, the task manager (section 3) is in charge of running all the necessary node modules, running also the perception module. Once perception module is called, the perception service can be called from a behavior tree node (explained in section 3), when it is necessary, in order to detect the cans to be grasped.



## 3 Task Manager

### 3.1 Introduction

The aim of this module is to manage all the tasks executed by the other modules as navigation, perception and arm's interface. Moreover, it manages other tasks that are necessary to perform all the entire process.

There are different ways to manage the different tasks of a system. The most direct way is to hard code it and manage all tasks by using loops and conditions. The problem with that comes when it is wanted to debug the code or add a new functionality. The bigger is the code, the more difficult it is to know what it is doing and to add new tasks.

The most feasible way to create a task manager is to do it in a modular way. Isolating the code of each task from the rest of code, but being possible to communicate among all tasks. This way, it is not a problem to add new tasks and the debugging problem disappears, because when something is not working quite well, it is possible to debug just the task that is not working without affecting the rest of tasks. Another advantage of being modular, is that the system can be built in different order, just ordering in a different way these modules, like black boxes, without having big problems.

### 3.2 Behavior Tree Library Theory

#### 3.2.1 Introduction

The behavior tree library allows to create different tasks, called nodes, that can be communicated among them by using ports. These nodes are controlled using controller nodes of different types, depending of the task's needs. In addition, there exist other type of control nodes, called decorator nodes, that allows to manipulate the flow of the behavior tree.

Each type of node, except *decorator nodes*, have three different type of states:

- RUNNING: still executing the node process.
- SUCCESS: the node has finished successfully.
- FAILURE: the node has finished without success.

Then, once all the nodes have been implemented using the behavior tree library [13], it is necessary to describe the behavior tree (BT). This is done with an XML file or using a GUI editor.

Groot is the GUI BT editor that has been chosen for the development of this project. Its own nomenclature will be used for the nodes explanation, i.e. the mentioned names of the type nodes come from the GUI editor Groot, despite behavior tree theory [17] uses different names in some cases.

### 3.2.2 Type of nodes

#### Control Nodes

This kind of nodes is in charge of executing their child nodes. There are different type of control nodes that condition the execution behavior of the rest of nodes. The different type of control nodes are the following:

- **Sequence Nodes**

A sequence node follows the AND logic. All its child nodes must return SUCCESS in order to finish the process successfully. If some of the children returns FAILURE, the process is going to be aborted and returns FAILURE.

In addition, sequence control node is composed of two different types:

- Sequence

- SequenceStar

The difference between these two types lies in the response when a child node returns a FAILURE. While *sequence* control node restarts again the sequence of nodes execution, *sequenceStar* control node ticks again the last executed node (i.e. it invokes the callback function of the last executed node), the one that returned a FAILURE.

- **Fallback Nodes** A fallback node follows the OR logic. At least its last child node must return SUCCESS in order to finish the process successfully. If the last child returns FAILURE, the process is also going to return a FAILURE. So, as it works as an OR logic, if one child returns a FAILURE it ticks the following child until all children have been ticked or a SUCCESS has been received from one child.

There are two types of fallback control nodes:

- Fallback

- FallbackStar

In this project only the standard fallback control node is used, because there are no tasks that requires to be interrupted in a asynchronous way as *fallbackStar* control node offers.

#### Decorator Nodes

Decorator nodes can only have one child and they are normally used for especial controls.

There are different types of decorator nodes:

- Inverter
- ForceSuccess
- ForceFailure

- Repeat
- RetryUntilSuccessful
- Timeout

### Action Nodes

Action nodes do not have any child node. It is necessary to be implemented in order to perform the necessary tasks.

### Condition Nodes

Condition nodes are very similar to action nodes. They must be implemented, but they should not return a RUNNING nor change anything on the system.

#### 3.2.3 Ports

As behavior tree nodes have a very similar concept a normal functions, they also use input parameters and output information (that at the same time can be used as input parameters by other behavior tree nodes). The use of ports can be very useful when there are some parameters that are required to be changed because of debugging reasons or because they can vary according to the scene or because some behavior tree nodes need to share some variables.

In this project, ports are not necessary because:

- All the parameters that are required to be changed, are set in a *launch* file.
- All the variables required to be shared among behavior tree nodes are shared with a class. So, an object from this class is passed to each behavior tree node when they are created.

### 3.3 Behavior Tree inside ROS

#### 3.3.1 How it works inside ROS

Using *BehaviorTreeCPP* all ROS functionalities continue working as normally, but now a new layer is added. So, instead of using different ROS nodes or functions for each step, an added layer managed by a behavior tree is used.

It can be seen that each layer has its own file. These files are the following ones:

- **node.cpp**: The called ROS node. It initializes the task manager.
- **behavior\_tree\_nodes.cpp**: Behavior tree manager nodes. Each behavior tree node calls a function from *Manager* class.
- **manager.cpp**: It contains a class, called *Manager*, with all the necessary functions to manage all the tasks that the robot should execute to perform all the process. All the robot functions that it uses comes from *Robot* class.

- **robot.cpp**: It contains a class, called *Robot*, with all the necessary functions to interact with the robot.

### 3.3.2 How to implement behavior tree nodes

#### Create a node

There are two ways to create a node:

- Creating a class by inheritance from class SyncActionNode as shown in listing 2.

Listing 2: BehaviorTree Node created by class inheritance.

---

```

1
2 class GoToServingTable: public SyncActionNode
3 {
4
5 public:
6     GoToServingTable(const std::string& name):
7         SyncActionNode(name, {})
8     {
9     }
10
11    BT::NodeStatus tick() override
12    {
13        std::cout << "GoToServingTable: " << this->name() << std::endl;
14        return BT::NodeStatus::SUCCESS;
15    }
16 };

```

---

- Creating a node using dependency injection with a function pointer, as shown in listing 3.

Listing 3: BehaviorTree Node created by function pointer.

---

```

1    BT::NodeStatus myFunction()

```

---

As all the functions that manage the tasks are inside *Manager* class it is necessary to insert an object of *Manager* class as an attribute in node's register.

Different ways to use a *Manager* object have been attempted:

- **Sharing the Manager class object using behavior tree ports.** The problem of this solution was the complexity of sharing each time the object and the low readability of this option. A new developer would not understand this data interchange.

- **Sharing the same Manager class object by using a pointer.** This was the chosen option. The object was shared defining an object pointer of class *Manager* in each inherited class, created to implement a behavior tree node. This way the same object was shared among all nodes in a more understood and readable way.

## Register a node

Once the nodes functionality are implemented it is time to register them for their later execution. There are two main ways.

- **Using a function or class method.**

In order to register a node with a function pointer it is necessary to indicate if it is an *action node* or a *condition node*, and it is possible to register it by using a function pointer or a method of a class. It is possible to see below how to register a node with each one of these options.

- Using a function pointer:

Listing 4: BehaviorTree Node created using a function pointer.

---

```

1 factory.registerSimpleAction("GoToSensingPosition",
2                               std::bind(GoToSensingPosition));
3 factory.registerSimpleCondition("CheckSensingPosition",
4                                 std::bind(CheckSensingPosition));

```

---

- Using a class method:

Listing 5: BehaviorTree Node created using a class method.

---

```

1 GripperInterface gripper;
2 factory.registerSimpleAction("OpenGripper",
3                             std::bind(&GripperInterface::open,
4                                     &gripper));
5 factory.registerSimpleCondition("CheckGripper",
6                                std::bind(&GripperInterface::check_gripper,
7                                         &gripper));

```

---

- **Using a node created by class inheritance.**

This is the most recommended option since it is not necessary to indicate which is the type of the node.

This was the chosen option but adding a parameter to the behavior tree node created by inheritance. This added parameter is an object pointer from class *Manager* used to execute all the necessary functions to perform all the tasks. Moreover, this object pointer shares with all behavior tree nodes a ROS node, being possible to execute different services and ROS functionalities without the need of creating each time a new node.

In order to have this object pointer inside the class, it is necessary to define the behavior tree class node as shown in listing 6 and register the node by using the same coding shape as shown in listing 7.

Listing 6: BehaviorTree Class Node with object pointer as parameter.

---

```

1 class OpenBar: public SyncActionNode
2 {
3
4     public:
5         // additional arguments passed to the constructor
6         OpenBar(const std::string& name, const NodeConfiguration& config,
7             Manager& manager):
8             SyncActionNode(name, config){_manager = &manager;}
9
10        NodeStatus tick() override;
11        static PortsList providedPorts() { return {}; }
12
13    private:
14        Manager *_manager;
15 };

```

---

Listing 7: BehaviorTree Node registration adding a object pointer as parameter.

---

```

1 NodeBuilder builderOpenBar = [&manager](const std::string& name,
2 const NodeConfiguration& config)
3 {
4     return std::unique_ptr<OpenBar>(<new> OpenBar(name, config, manager));
5 };
6 TreeNodeManifest manifestOpenBar =
7     BehaviorTreeFactory::buildManifest<OpenBar>("openBar");
8 factory.registerBuilder(manifestOpenBar, builderOpenBar);

```

---

### 3.3.3 How to build the logic process

Once the nodes behavior have been implemented and registered, it is time to build the logic process that the entire system is going to follow to perform all the tasks.

Below is described how to do it with an XML file and with Groot.

#### XML file

In BehaviorTree library documentation [18] the format that the XML file must follow is well explained. In appendix B the XML file of this project is shown. It was created using Groot.

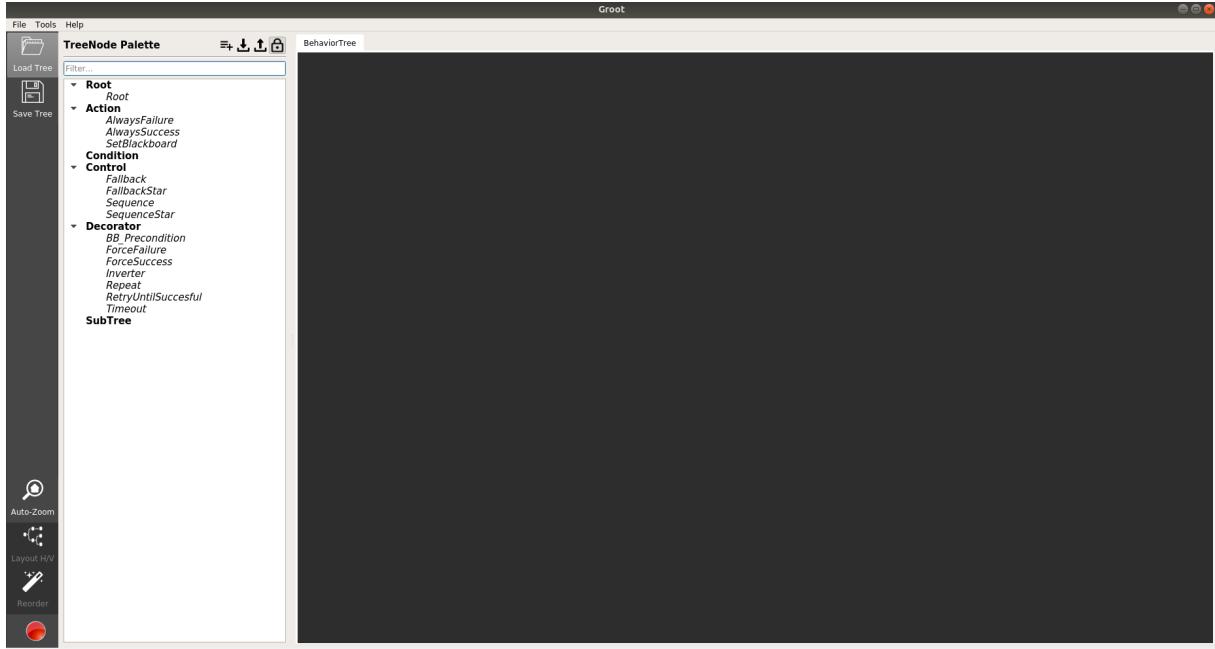


Figure 23: Blank Groot interface.

## Groot

Groot is a GUI that allows to build the XML that defines the logic process graphically. In figure 23 the Groot editor and its *TreeNode Palette* are shown, with the different default nodes that the Groot editor offers to build a behavior tree. They are the following:

- **Root**

- Root → It is a necessary node. Without this node at the beginning of the behavior tree, the XML file is not created.

- **Action** All action nodes are registered here. The default action nodes are the following ones:

- *AlwaysFailure* → Creates a node without any action that always returns a failure. Normally used for debugging reasons.
- *AlwaysSuccess* → Creates a node without any action that always returns a success. Normally used for debugging reasons.
- *SetBlackBoard* → Set a value for a port. Normally used for debugging purposes.

- **Control**

- *Fallback* → Implements fallback control node.
- *FallbackStar* → Implements fallback star control node.

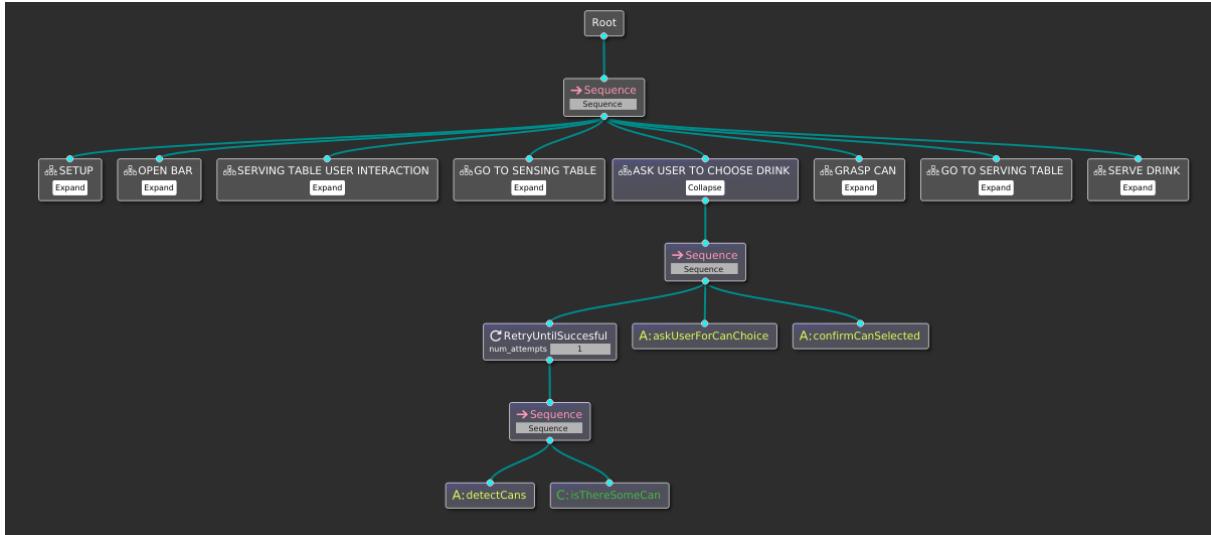


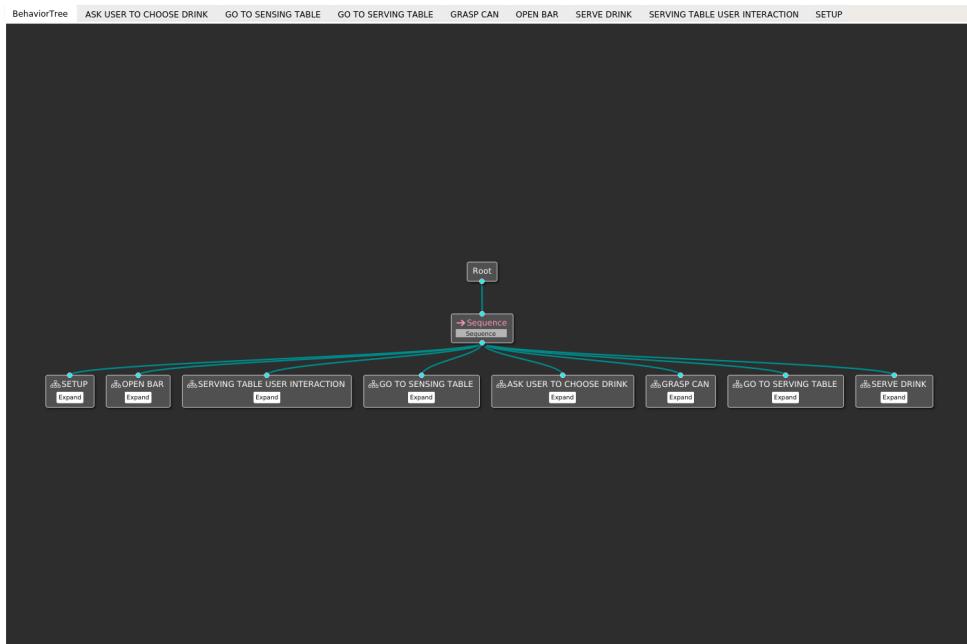
Figure 24: Blank Groot interface.

- *Sequence* → Implements sequence control node.
- *SequenceStar* → Implements sequence star control node.
- **Decorator** Nodes with just one child, that apply special effects to its child nodes.
  - *BB\_Precondition* → Node used as conditional, that returns a success when condition is fulfilled.
  - *ForceFailure* → Forces the failure of a node. Normally used when a failure is required for debugging reasons.
  - *ForceSuccess* → Forces the success of a node. Normally used when a success is required for debugging reasons.
- **SubTree** In this section all sub-trees are listed. The sub-trees are groups of nodes with a bigger functionality. As an example, Figure 24 shows that to give the possibility to the user to choose a drink, a sequence of tasks must be executed. Therefore, the sequence of actions to allow user choose drink is grouped in a sub tree-called *ASK USER TO CHOOSE DRINK*.

Figure 25 shows the same editor with the created action and condition nodes, in addition to created sub-trees.

### 3.4 Structure used in this project

Taking into account previous sections about behavior tree, the project's behavior tree will be explained and developed in this section.



(a) Graphical Behavior Tree.

- ▼ **Condition**
  - isCanGrasped
  - isGoToSensingTableSolved
  - isGoToServingTableSolved
  - isPreviousDrinkServedCorrectly
  - isRobotLocalizedSensingTable
  - isRobotLocalizedServingTable
  - isThereSomeCan
  - isValidPath
- ▼ **Root**
  - Root*
- ▼ **Action**
  - AlwaysFailure
  - AlwaysSuccess
  - SetBlackboard
  - askForDrinks
  - askUserForCanChoice
  - askUserToSolveProblem
  - computePath
  - confirmCanSelected
  - detectCans
  - goToSensingTable
  - goToSensingTableFailure
  - goToServingTable
  - goToServingTableEnd
  - headTowardUser
  - initNodes
  - initializeVariables
  - openBar
  - prepareToGoToSensingTable
  - prepareToGoToServingTable
  - prepareToGraspCan
  - readParameters
  - serveDrink
  - transformStamped
  - tryToGraspCan
- ▼ **Control**
  - Fallback
  - FallbackStar
  - Sequence
  - SequenceStar
- ▼ **Decorator**
  - BB\_Precondition
  - ForceFailure
  - ForceSuccess
  - Inverter
  - Repeat
  - RetryUntilSuccessful
  - Timeout
- ▼ **SubTree**
  - ASK USER TO CHOOSE DRINK
  - GO TO SENSING TABLE
  - GO TO SERVING TABLE
  - GRASP CAN
  - OPEN BAR
  - SERVE DRINK
  - SERVING TABLE USER INTERAC...
  - SETUP

(b) Root and Action nodes.

(c) Condition, Control and Decorator nodes and SubTree.

Figure 25: System Behavior Tree created in Groot GUI.

### 3.4.1 General view

Figure 26 shows the general view of the entire system process of this project. The process is made by a sequence of different sub-trees:

- Open bar

TIAGo is introduced to the user and starts at its initial position.

- Serving table user interaction

TIAGo goes to user's table and asks if the user wants a drink.

- Go to sensing table

TIAGo starts to locate and navigate to the sensing table.

- Ask user to choose drink

TIAGo detects the drinks, sends the image with number labeled drinks and asks user to choose one.

- Grasp can

TIAGo grasps the can.

- Go to serving table

TIAGo locates and navigates to the serving table with the selected drink in hand.

- Serve drink

TIAGo serves drink to user and throws the can to a recycle bin.

### 3.4.2 System process explanation

Each of the sub trees is made by different nodes. This is shown below.

#### **Open bar**

Inside open bar sub-tree, shown in figure 27, there is just one node called open bar. This node is in charge of introducing TIAGo with a voice message, move until the initial joint positions and disable head manager of TIAGo in order to control the head without undesired movements. This tree can be seen expanded in figure 27.

#### **Serving table user interaction**

Inside this sub-tree, shown in figure 28, a sequence of 2 nodes is executed.

- goToServingTable: The robot navigates until a position in front of serving table, where the user is.

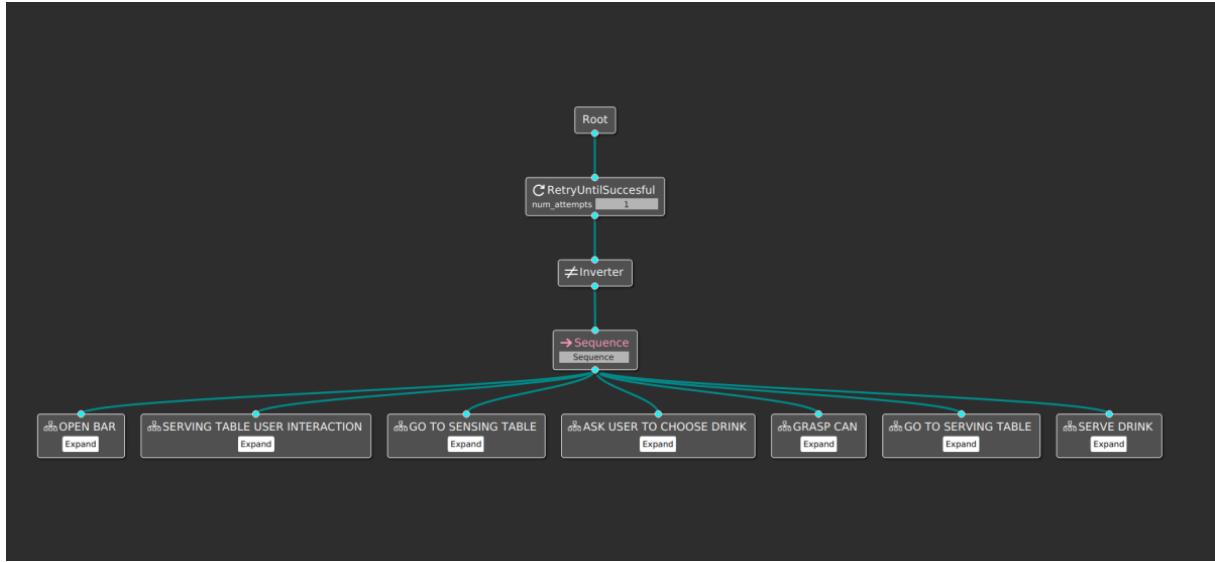


Figure 26: Project's behavior tree general view with sub-trees.

- askForDrinks: The robot asks the user if he/she wants a drink. Then the user must answer using the terminal.

As both nodes are executed by a *sequence* control node, until both action nodes are not completed, it is not possible to pass to the next sub-tree.

### Go to sensing table

Inside this sub-tree, shown in Figure 29, there are five nodes controlled by different control nodes. This sub-tree follows the next procedure in a sequence control node, i.e. two branches connected to main sequence should return a SUCCESS in order to complete the sub-tree successfully.

- A first branch connected to sequence control node.

In the first branch there is a loop that retries the child node process during the indicated attempts or until receiving success. The child of this loop node is a **fallback**, that means that if one of its children returns a SUCCESS, the rest of nodes are not necessary to be executed and it returns a SUCCESS finishing in this way the loop.

The first child is a sequence that executes the node *prepareToGoToSensingTable* and a loop of a sequence control node that executes nodes *goToSensingTable* and *isRobotLocalizedSensingTable*.

- **prepareToGoToSensingTable** is a node that makes the robot to execute the necessary movements to go to the sensing table.
- **goToSensingTable** is a node that allows to move the robot to the position in front of sensing table.

- **isRobotLocalizedSensingTable** is a node that checks if the robot is in the correct position, in front of sensing table.

On the other hand, the second child of fallback control node is a condition called *isGoToSensingTableSolved* connected to an inverter.

- **isGoToSensingTableSolved** is a node that checks if the robot can retry again the action of going to sensing table. Returns SUCCESS when it is possible to retry again to go to sensing table, so with the inverter it returns a FAILURE in order to retry again the process marked by the fallback control node.

- A second branch is connected to a sequence control node just to check if the robot is in the position in front of sensing table.

### **Ask user to choose drink**

Inside this sub-tree, shown in Figure 30, there is a sequence control node that executes three branches connected to it.

- A loop that executes a control node sequence until SUCCESS or number of attempts used. This sequence executes two nodes.
  - **detectCans** This node is in charge of detecting the drinks that are on the sensing table and label them with numbers to show them to the user.
  - **isThereSomeCan** This node checks if there is some can detected. If there is some detected can, the sequence control node is completed successfully, otherwise sequence control node is executed again if there are still attempts to execute it.
- **askUserForCanChoice** This node asks the user for the drink that he/she wants to select.
- **confirmCanSelected** This node makes the robot say something to the user in order to confirm its drink selection.

### **Grasp can**

Inside this sub-tree, shown in figure 31, the robot is prepared to grasp the selected can and then grasp it. It follows the next procedure:

- In the first branch connected to main sequence control node, the robot performs all the necessary movements to be prepared to grasp a can.
- It computes the *inverse kinematics* to make the end effector reach the chosen can. In case it is not possible to obtain an available *inverse kinematics* the robot corrects its position in order to try it again. This two actions are performed until compute good *inverse kinematics* or expend some limited number of attempts.

- Once *inverse kinematics* are computed it is time to make the robot be prepared to compute a path to reach the chosen can. Normally, this preparation is to have a good arm configuration to obtain a simpler path.
- Then, the path is computed and, as happens with the computation of inverse kinematics, if there is not a feasible path, the position of the robot is corrected and then the path computation is tried again until success or a limited number of attempts are spent.
- Finally, with the path obtained, the robot tries to grasp the can.

### Go to serving table

Inside this sub-tree, shown in Figure 32, there are five nodes controlled by different control nodes. This sub-tree follows the next procedure in a sequence, i.e. the two branches connected to main sequence should return a SUCCESS in order to complete the sub tree successfully.

- A first branch connected to a sequence control node.

In the first branch there is a loop that retries the child node process during the indicated attempts or until receiving success. The child of this loop node is a **fallback**, that means that if one of its children returns a SUCCESS, the rest of nodes are not necessary to be executed and it returns a SUCCESS finishing in this way the loop.

The first child is a sequence that executes the node *prepareToGoToSensingTable* and a loop of a sequence control node that executes nodes *goToSensingTable* and *isRobotLocalizedSensingTable* until get a SUCCESS or not having more attempts.

- **prepareToGoToServingTable** is a node that makes the robot to execute the necessary movements to go to the serving table.
- **goToServingTableEnd** is a node that allows to move the robot to the position in front of sensing table.
- **isRobotLocalizedServingTable** is a node that checks if the robot is in the correct position, in front of serving table.

On the other hand, the second child of fallback control node is a condition called *isGoToServingTableSolved* connected to an inverter.

- **isGoToServingTableSolved** is a node that checks if the robot can retry again the action of going to serving table. Returns SUCCESS when it is possible to retry again to go to serving table, so with the inverter it returns a FAILURE in order to retry again the process marked by fallback control node.
- A second branch connected to a sequence control node just to check if the robot is in the position in front of serving table.

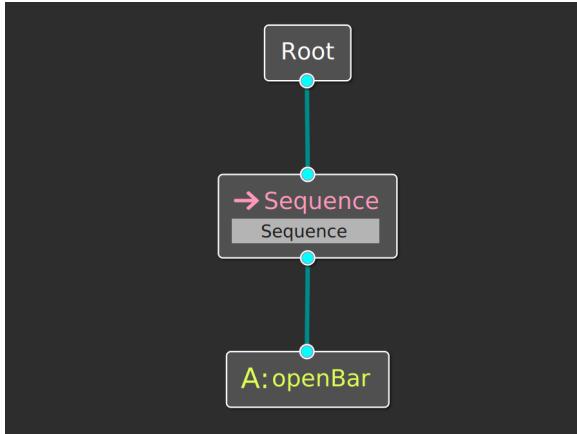


Figure 27: Open bar.

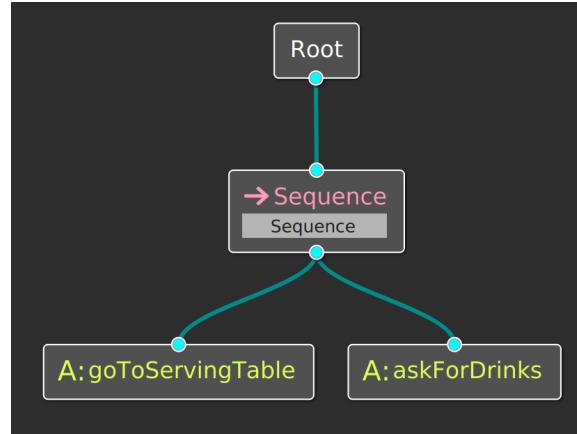


Figure 28: Serving table user interaction.

### Serve drink

Inside this sub-tree, shown in figure 33, there is just an action node called **serveDrink** that makes the robot to serve the selected drink to the user in a glass located on serving table. Then, the drink can is thrown to the recycle bin next to the serving table and TIAGO executes the node *closeBar* performing all the necessary movements to finish the entire system process. Finally, it starts again.

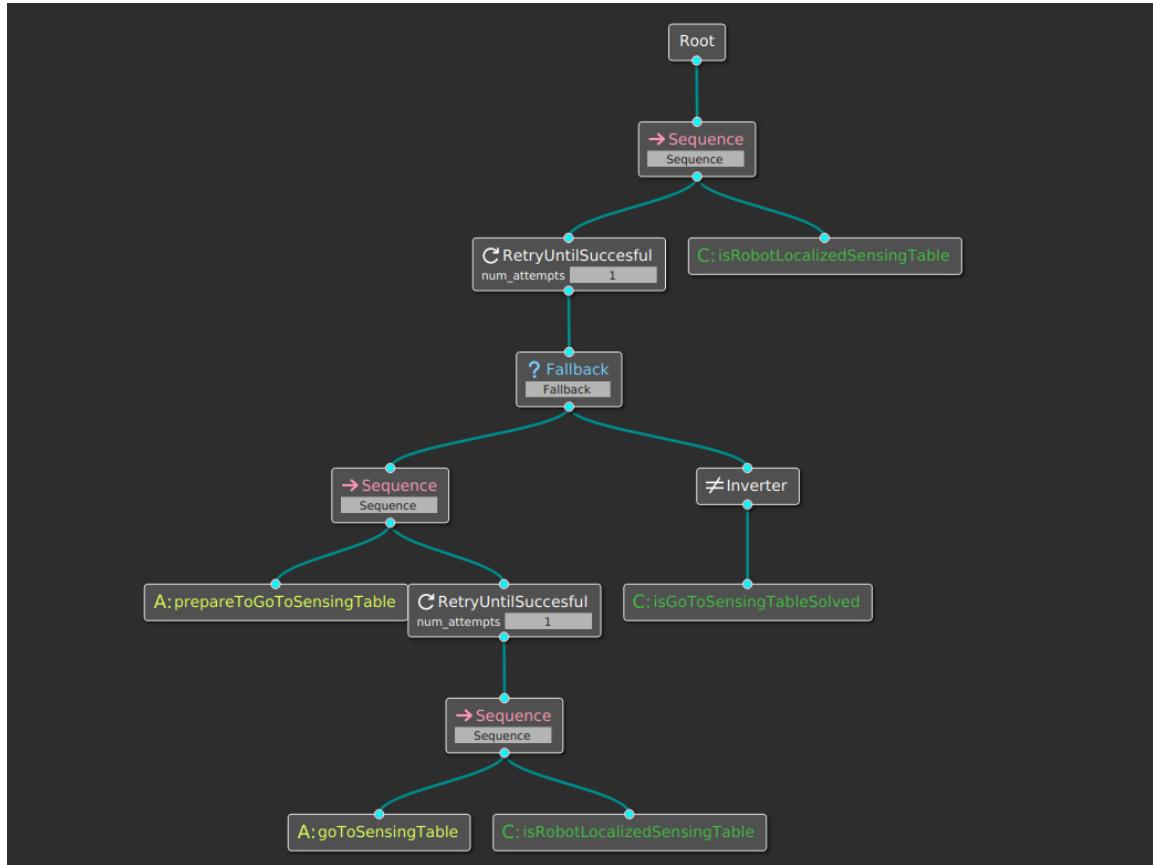


Figure 29: Go to sensing table.

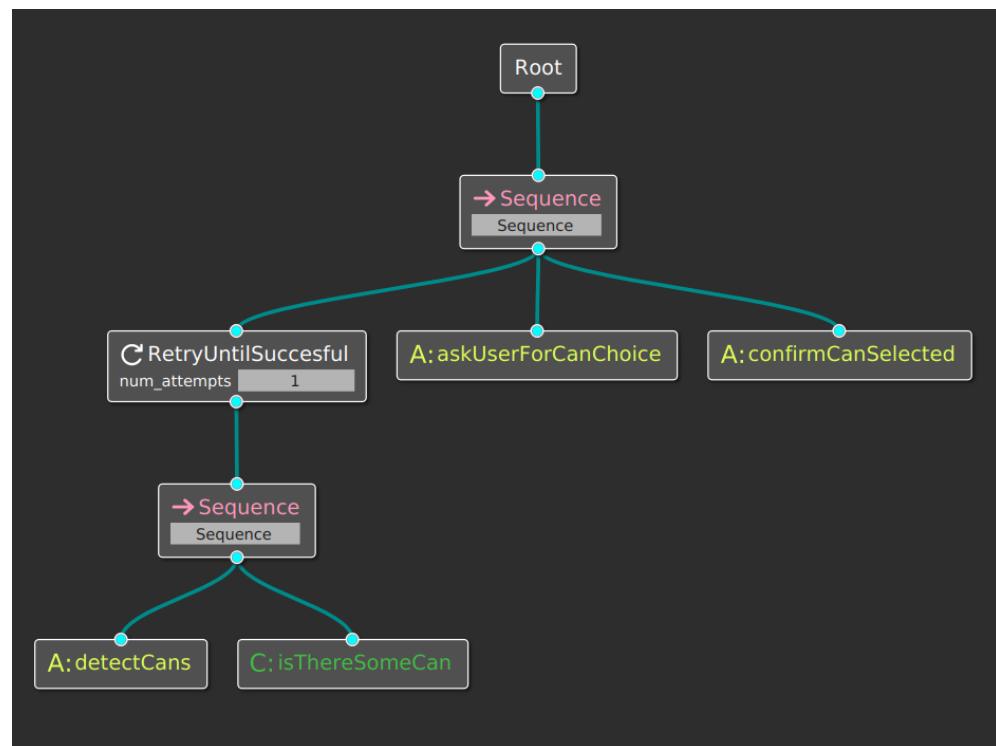


Figure 30: Ask user to choose drink.

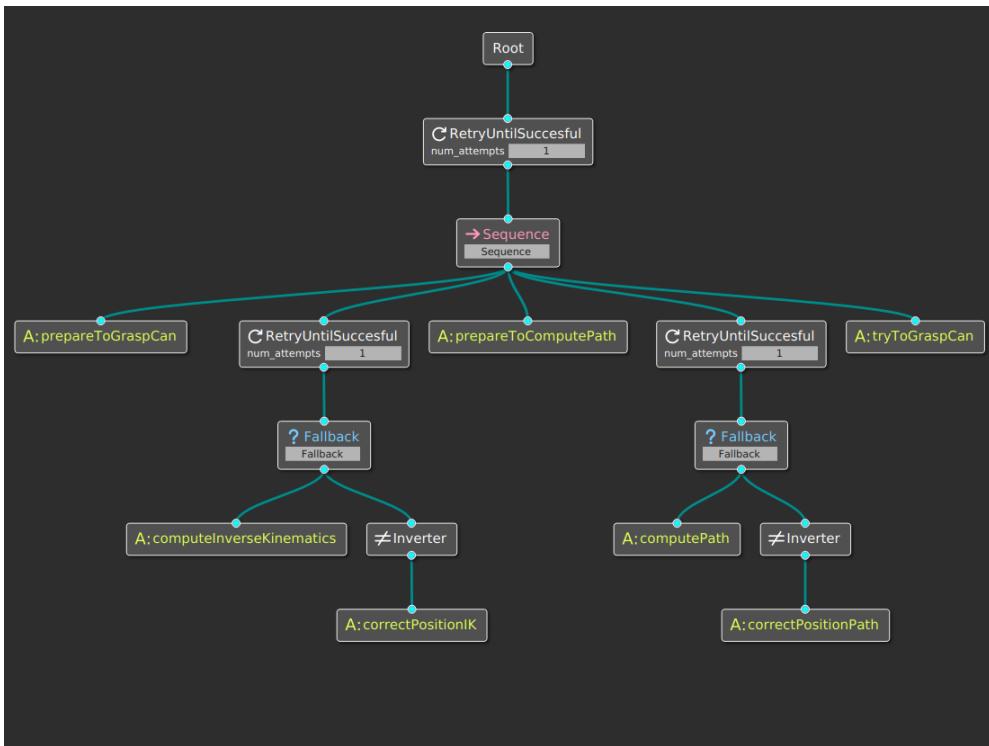


Figure 31: Grasp can.

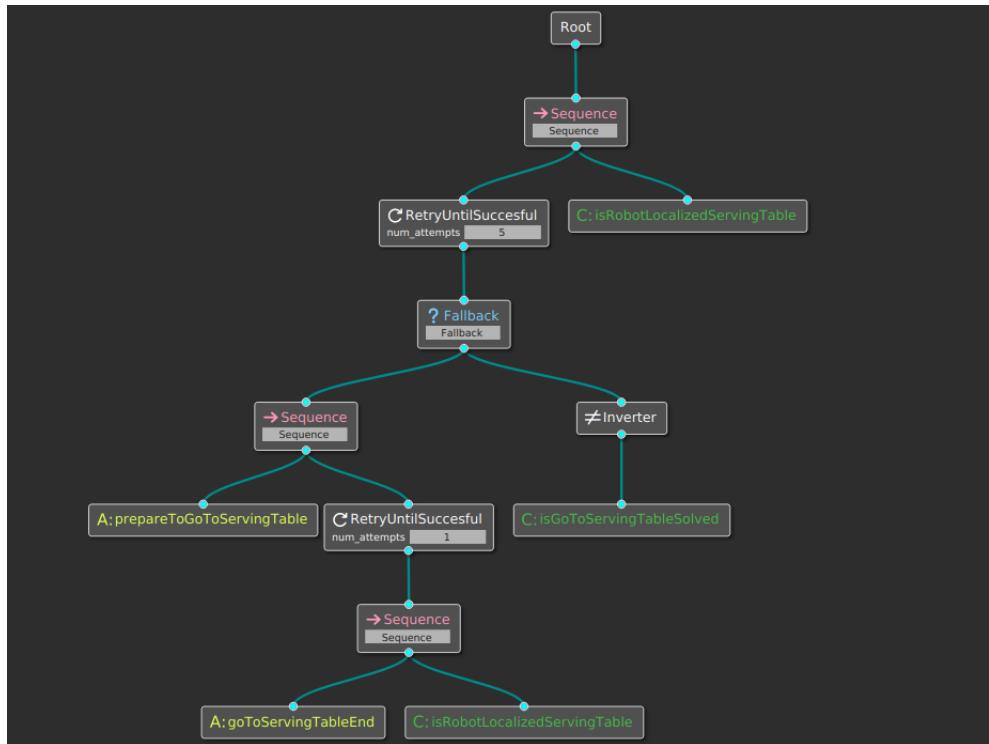


Figure 32: Go to serving table.

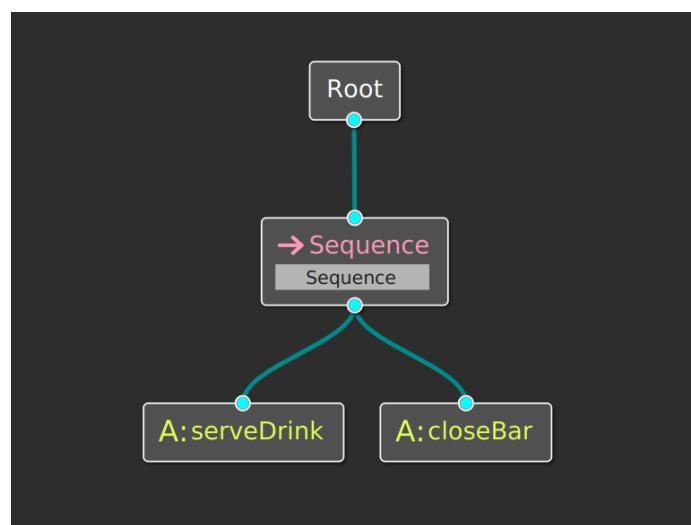


Figure 33: Serving drink.



## 4 Experiments and Results

In this section, the different experiments performed for the perception and task manager modules and the general process will be explained. The different results obtained from these experiments are used to analyze the behavior of the solutions.

Some quantitative and qualitative data are analyzed for the perception part. On the other hand, for the task manager part, it is just necessary to perform some experiments to obtain qualitative data.

A video was used to perform an experiment of the task manager. That is why it is important to remark that in the Appendix A there is a link to a code repository page that, aside from all the code used in these tests, includes several videos of the different experiments mentioned in this section, both in this document and in that one from my partner with the parts of navigation and planning.

### 4.1 Perception experiments

Perception was tested checking whether the position and orientation of the soda cans, computed by pose estimation algorithm, has a small enough error or not. In order to check that, three experiment were performed.

- **Check orientation** With this experiment the orientation of the cans is evaluated. Three samples were used, consisting of nine soda cans each.

In order to make it easy to check this orientation, Markers in rviz were used. This way it is possible to check it in a visually. Therefore, for performing this experiment, the perception module should be run. The perception module detects the soda cans and estimates their position, then publishes in a ROS topic the position and orientation using a *MarkerArray* message. Finally, it is possible to compare the orientation of each labeled can, shown in the capture done by perception module, with the orientation displayed with the rviz markers.

For the purpose of comparing both orientations it is necessary to take into account the direction of each axis for each can:

- **X-axis** it is shown in red in rviz markers, and must go from the center of the can towards the hole of the can.
- **Y-axis** it is shown in green in rviz markers, and it is perpendicular to X axis in top of the can's plane.
- **Z-axis** it is shown in blue in rviz markers, and it is normal to the X and Y axis.

As a result of this experiment, Figures 34, 35 and 36 compare the result obtained with *pose estimation* algorithm with the real orientation.

For each figure, left and right sub-figures have a similar perspective view, so it is easy to check that the orientation of the cans is correct for the first two samples.

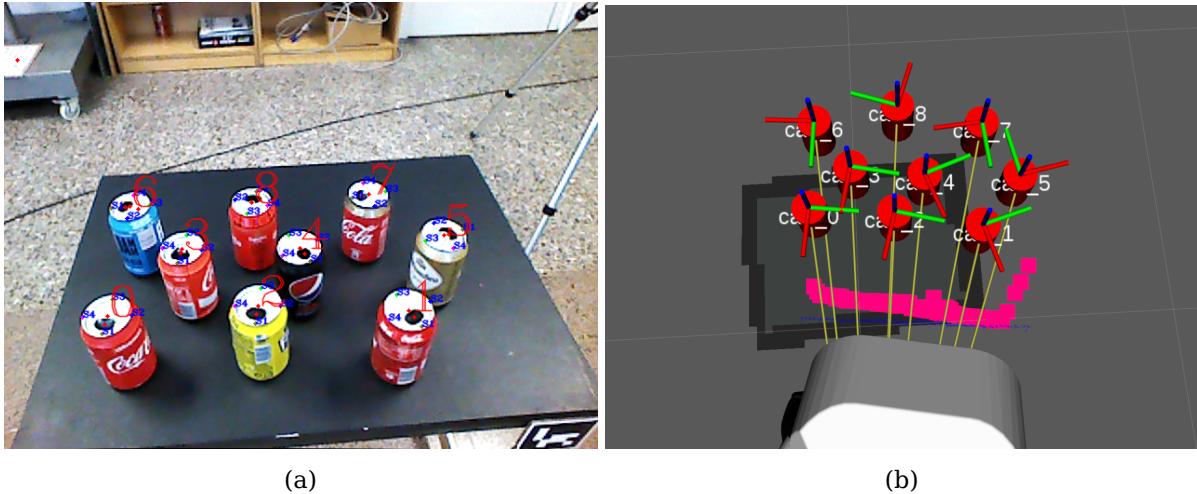


Figure 34: Experiment to check orientation with sample 1. A ROS topic was used to publish a message of type [visualization\\_msgs/MarkerArray](#).

On the other hand, there is a mistake with can number 6, from last sample. Due to the position where it is, it is more probable to have some detection mistakes. In this case, the center of the hole was not found with accuracy and the angle formed by the center of the can and the hole creates a wrong  $S_1$  point, and in consequence, a wrong x-axis direction and wrong orientation of the can.

- **Aruco markers comparison** With this experiment we want to compare the positions given by the perception module with the positions given by a typical and very used pose estimation algorithm, as the Aruco markers detector.

In order to execute this experiment, the following steps are performed:

- 9 cans of soda are placed on the sensing table.
- These 9 cans are detected with the perception algorithm and their pose estimation is returned.
- The returned pose estimation is published as a marker in order to see it on rviz and it is also possible to see the positions with the values returned by the service `tiago_sensing/detectObjects`.
- It is placed an aruco at each can and their position is obtained from the pose topic published by aruco detector algorithm. It is also possible to see this pose in rviz and compare both positions graphically.
- Finally, the values are compared and an error is obtained.

For this experiment, two samples will be used, sample 1 (the same as the last one used in the previous experiment) and sample 2 shown in figure 37. In order to compare the positions obtained with perception module and the ones obtained with aruco detector it is better to use quantitative data instead of showing the results graphically.

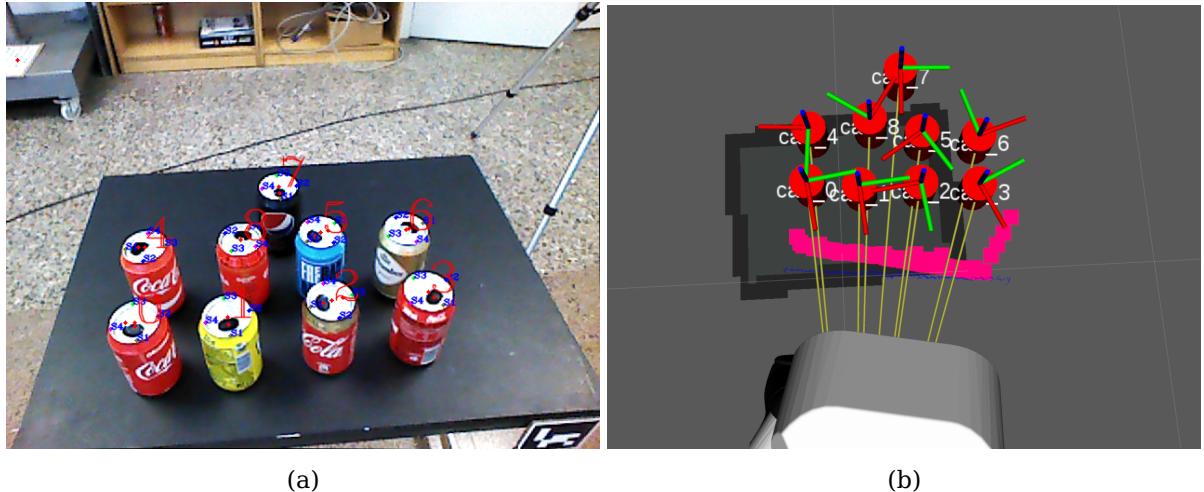


Figure 35: Experiment to check orientation with sample 2. A ROS topic was used to publish a message of type `visualization_msgs/MarkerArray`.

In tables 1 and 2 can be seen the results obtained from the position comparison between the position obtained using perception module and the one obtained applying aruco detector module for sample 1 and sample 2 respectively. In that table can be observed that the error is minimal, normally it is not higher than 2 cm both in X and Y axis. But this error is critical to grasp the user's chosen can or not.

It should be taken into account that it is possible to get some position error with aruco markers, because of a little bad detection or, because of human error when placing the aruco marker on the top of the can. The center of aruco may not be aligned with the center of the top of the can.

- **End effector position comparison**

On the other hand, the position of the cans of soda obtained from the perception algorithm, are compared with the position of the end effector. This way, it is possible to check if the real positions of the end effector and the cans correspond with the positions shown in rviz. This is useful to check if the path planner will be able to reach to can's position before executing the movement, just moving by hand the end effector to the can's position.

It is necessary to follow steps below to perform this experiment:

- Two cans of soda are placed on the sensing table.
- These two cans are detected with the perception algorithm and their pose estimation is returned.
- The returned pose estimation is published as a marker in order to see it on RVIZ and it is also possible to see the positions with the values returned by the service `tiago_sensing/detectObjects`.

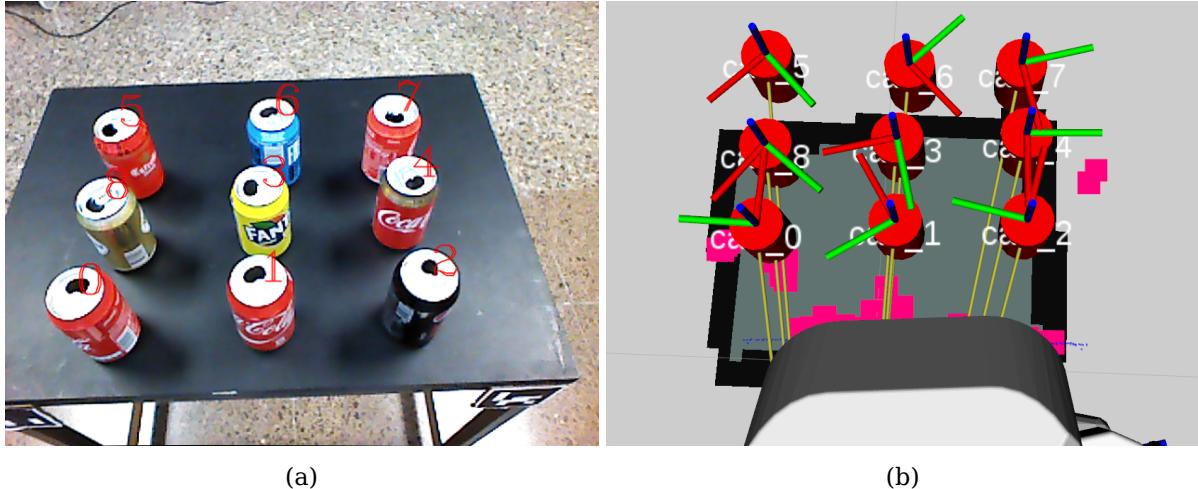


Figure 36: Experiment to check orientation with sample 1. A ROS topic was used to publish a message of type [visualization\\_msgs/MarkerArray](#).

Can	Module	X	Y	X error	Y error
0	Perception	0,4436	0,2455	-0,0066	0,0045
	Aruco	0,4370	0,2500		
1	Perception	0,4578	0,0658	-0,0062	0,0053
	Aruco	0,4516	0,0711		
2	Perception	0,4671	-0,1056	-0,0071	0,0026
	Aruco	0,4600	-0,1030		
3	Perception	0,5686	0,0712	0,0014	0,0088
	Aruco	0,5700	0,0800		
4	Perception	0,5840	-0,1014	0,0061	0,0028
	Aruco	0,5901	-0,0986		
5	Perception	0,6778	0,2576	-0,0004	0,0146
	Aruco	0,6774	0,2722		
6	Perception	0,6736	0,0651	0,0124	0,0049
	Aruco	0,6860	0,0700		
7	Perception	0,6827	-0,0913	0,0181	0,0057
	Aruco	0,7008	-0,0856		
8	Perception	0,5512	0,2453	0,0174	0,0217
	Aruco	0,5686	0,2670		

Table 1: These errors are computed for sample 1. They are obtained by comparing the position of the soda cans obtained from perception module with the position obtained from aruco detector.

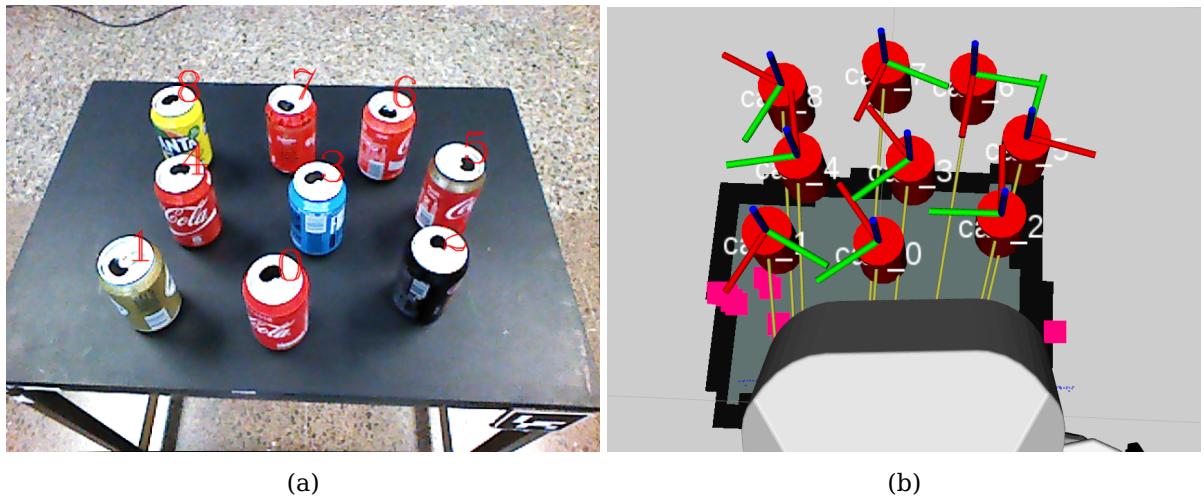


Figure 37: Experiment to check orientation with sample 2. A ROS topic was used to publish a message of type [visualization\\_msgs/MarkerArray](#).

Can	Module	X	Y	X error	Y error
0	Perception	0,4601	0,0791	-0,0051	0,0019
	Aruco	0,4550	0,0810		
1	Perception	0,4655	0,2351	-0,0015	0,0110
	Aruco	0,4640	0,2461		
2	Perception	0,5092	-0,0894	-0,0005	0,0078
	Aruco	0,5087	-0,0816		
3	Perception	0,5812	0,0451	0,0022	0,0018
	Aruco	0,5834	0,0469		
4	Perception	0,5768	0,2056	0,0030	0,0124
	Aruco	0,5798	0,2180		
5	Perception	0,6254	-0,1190	0,0092	-0,0095
	Aruco	0,6346	-0,1285		
6	Perception	0,7198	-0,0388	0,0029	0,0057
	Aruco	0,7227	-0,0331		
7	Perception	0,7238	0,0917	0,0038	0,0100
	Aruco	0,7276	0,1017		
8	Perception	0,6860	0,2374	0,0185	0,0159
	Aruco	0,7045	0,2533		

Table 2: These errors are computed for sample 2. They are obtained by comparing the position of the soda cans obtained from perception module with the position obtained from aruco detector.

- The gravity compensation feature of TIAGo robot is enabled and the end effector of the robot is moved by hand simulating the purpose of grasping a can.
- For each can, the end effector is moved until the can is in between the tool fingers.
- Finally, it is possible to see graphically if with the position obtained by the perception module, the robot would be able to grasp the can or not, applying correctly the planner to that position.

As it can be observed in figures 38 and 39, when the movement of the grasping tool to the chosen cans is simulated and it is in the good position to grasp the can, it is seen that the can fits in between grasping tool fingers, so the real position coincides with the position seen in rviz. But there is a case, in figure 39d, where the can does not fit with the grasping tool due to a failure in pose estimation.

## 4.2 Task manager experiments

- **Behavior tree checked with terminal messages**

The first experiment that must be done in order to check if the behavior tree built with Groot works well or not, it is printing messages that indicate which behavior tree node is running at the moment. This way it is possible to see in which node it is the program at each moment and it is possible to change some return values in order to check if everything is working well or not.

Two tests were performed for this experiment:

- For the first test, all the executed nodes must return a SUCCESS, simulating that everything goes well. This behavior is shown in Figure 40a, where it can be observed which nodes have been executed printing their names.
- For the last test, the navigation towards sensing table was forced to fail, that is why in Figure 40b can be observed that some messages are repeated. These messages show the FAILURE state received by *isRobotLocalizedSensingTable* at the bottom of the behavior tree (Figure 41). Moreover, two loop control nodes have been added. The first one, makes the fallback control node be repeated until success or two attempts, and the other one makes the sequence control node, that executes *goToSensingTable* and *isRobotLocalizedSensingTable* nodes, to be repeated until success or three attempts.

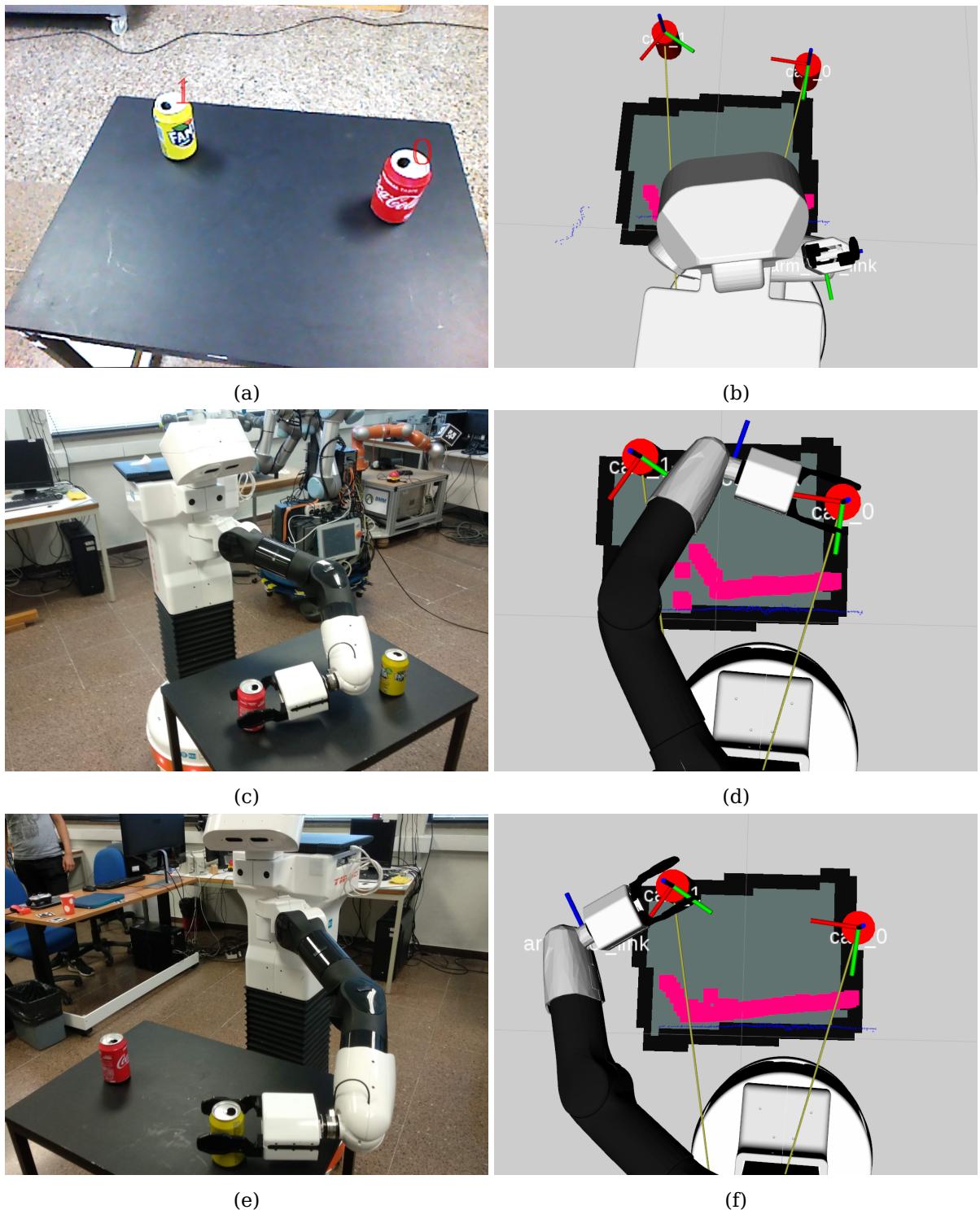


Figure 38: Experiment with sample 1. It shows if moving the grasping tool of TIAGo, it is possible to make the can fit between grasping fingers with the position of each can given by perception module.

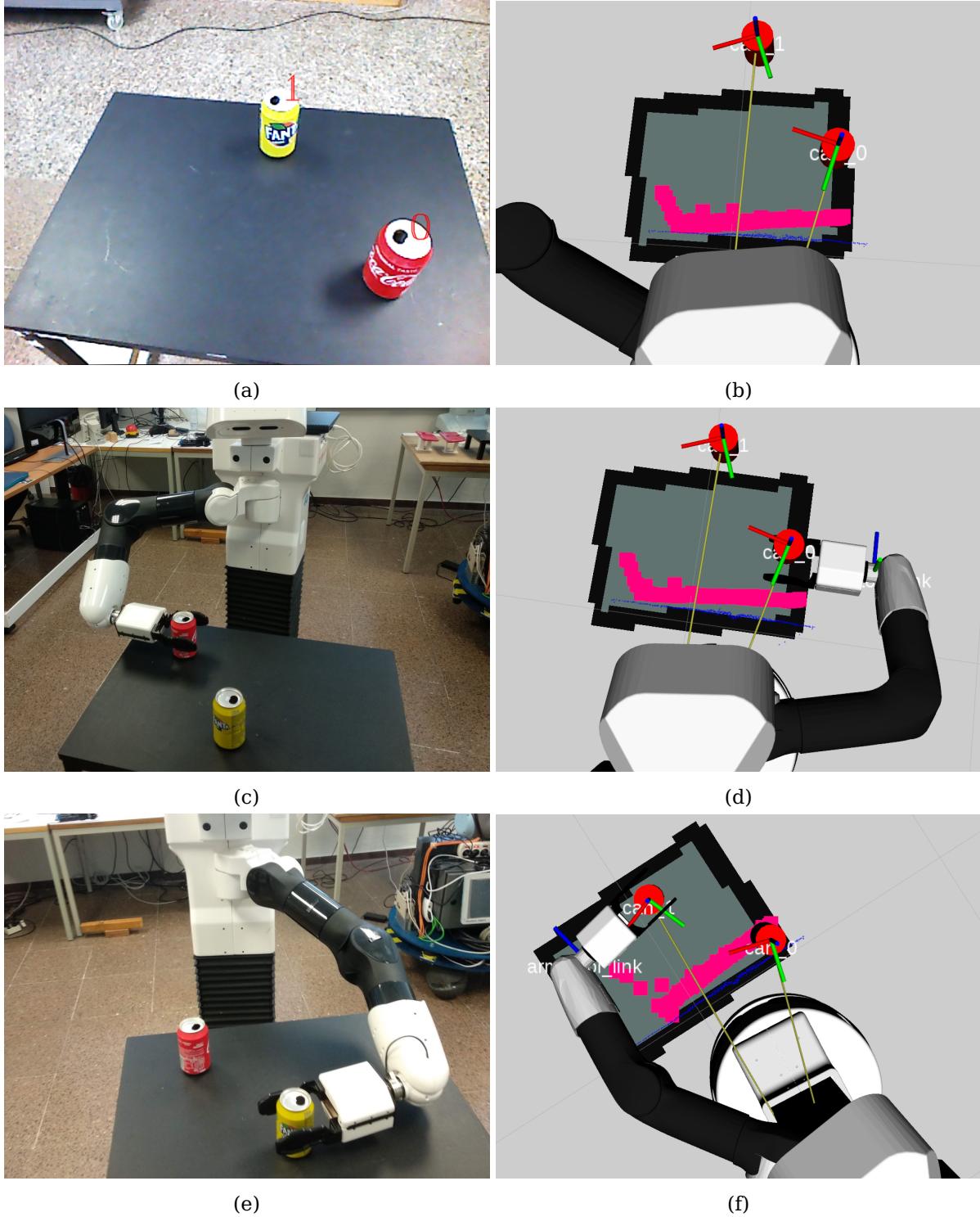


Figure 39: Experiment with sample 2. It shows if moving the grasping tool of TIAGo, it is possible to make the can fit between grasping fingers with the position of each can given by perception module.

```
openBar
goToServingTable
askForDrinks
prepareToGoToSensingTable
goToSensingTable
isRobotLocalizedSensingTable
isRobotLocalizedSensingTable
detectCans
isThereSomeCan
askUserForCanChoice
confirmCanSelected
prepareToGraspCan
computeInverseKinematics
prepareToComputePath
computePath
tryToGraspCan
prepareToGoToServingTable
goToServingTableEnd
isRobotLocalizedServingTable
isRobotLocalizedServingTable
serveDrink
closeBar
```

```
openBar
goToServingTable
askForDrinks
prepareToGoToSensingTable
goToSensingTable
isRobotLocalizedSensingTable
goToSensingTable
isRobotLocalizedSensingTable
goToSensingTable
isRobotLocalizedSensingTable
isGoToSensingTableSolved
prepareToGoToSensingTable
goToSensingTable
isRobotLocalizedSensingTable
goToSensingTable
isRobotLocalizedSensingTable
goToSensingTable
isRobotLocalizedSensingTable
isGoToSensingTableSolved
```

(a)

(b)

Figure 40: Output printed in terminal for each experiment.



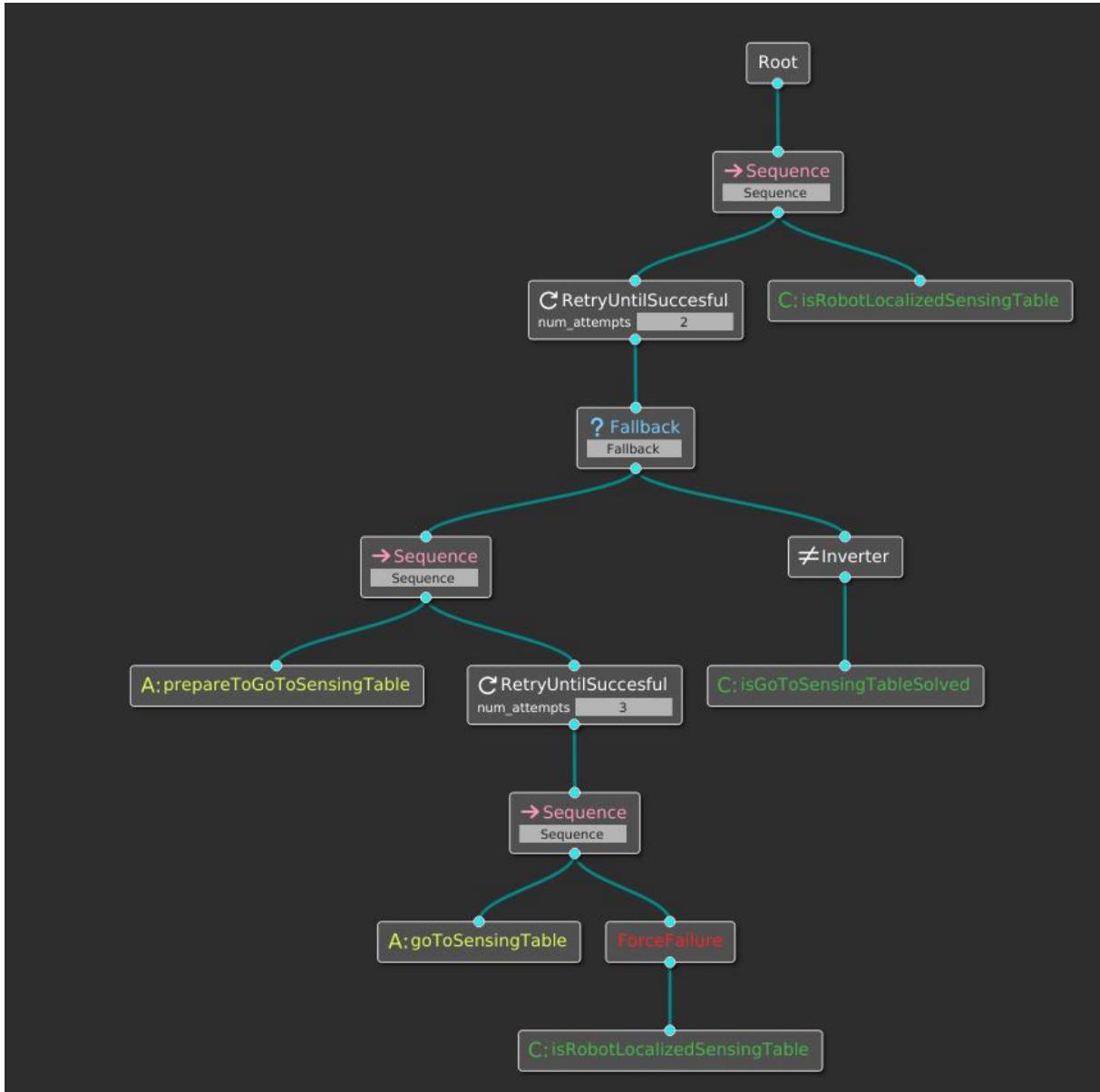


Figure 41: Behavior tree used in *goToSensingTable* sub-tree modified in order to force a failure.

- **Experiment with real robot movements**

In this experiment we want to show the capabilities of the behavior tree in the real platform. In order to do that, a video with the execution of some of the nodes of the behavior tree have been recorded. This video can be watched [here](#).

In this video it is possible to see the execution behavior of the three first sub-trees of the system:

- At minute **0:18** *openBar* node starts with TIAGo saying "hello" and offering a drink to the user looking approximately to his/her direction.
- At minute **0:41** *goToServingTable* node starts and TIAGo goes to the serving table.
- At minute **1:14** TIAGo asks the user if he/she wants a drink executing node *askForDrinks*
- At minute **1:20** TIAGo is preparing to go to sensing table applying *prepareToGoToSensingTable* node.
- Finally, at minute **1:32** it goes to sensing table applying *goToSensingTable* and *isRobotLocalizedSensingTable* nodes.

With this experiment it can be observed that the designed behavior tree can execute ROS functions in real robot without any problem.

### 4.3 General process experiments

Once all the different blocks in which the project was structured were proven to work correctly to fulfill their parts, different tests in order to complete the full waiter process were planned. The idea was to test how all the modules worked together and see how efficient and robust the method was.

The first step was to try to complete the most simple waiter process possible and test if the solution worked and what was its performance. This scenario was a very simple test with just one can in the picking table and with a full knowledge a priory of the position of the glass in the serving table. Videos of the different tests performed are included in the web page to appreciate how the solution works and how it responded to these experiments.

After this first simple test, the process was to keep on increasing complexity in order to try to find the black spots of the method to be able to design strategies to solve them. For time lack reasons and other complications explained in the conclusion chapter, the complexity of the tests could not be increased much and the videos presented show scenarios not too complex.

The results of these tests are very different and there are many problems that came out when performing them. The reasons behind these problems as well as more discussions on how to tackle them are included in section [8](#).



## 5 User manual

There are different steps that the user interfacing the program must execute in order to reproduce the entire system procedure. This section will explain all the required actions that must be performed just before executing the program and its setup, afterwards the interactions that the user must do with *TIAGo* in order to get a drink will be detailed step by step.

### 5.1 Required actions before execution the entire system process

These are the steps that should be followed in order to get all the requirements before using *TIAGo* as waiter.

#### 5.1.1 Environment setup

##### Picking or Sensing table

The *picking or sensing table* is the table used to put on the drinks that are going to be detected and grasped by the robot.

**Physical characteristics:** This table must fulfill some requirements to make easier the drinks detection.

- The **table** should be **black** in order to avoid light brightness and affect negatively to the image processing algorithm.
- For the same reason, the table should stay on non brightness floor.
- Its height should be according to the height capabilities of *TIAGo*, specially, taking into account *torso joint* limits.

##### Position

The table should be in some position with wide space, in order to ease the collision-free navigation of *TIAGo*.

##### Drinks location

Drink cans should be allocated on the table inside a limited zone to avoid confusion between the top of the cans and the background.

##### Serving table

The serving table is the location where the user has its glass. The user will use this glass to drink, so it will be the glass where *TIAGo* must serve the drink chosen by the user.

##### Location

The serving table should be in a location a bit away from the other table with a wide space to allow *TIAGo* move without problems its arm. It can be modified by changing the *parameters.launch* file.

## Serving glass

The serving glass should be wide enough, approximately 8 cm of diameter in the tops side, in order to have more error margin to serve the drinks. The position of the glass is set in the transformation tree of the system.

## Recycle bin

There is a recycle bin where *TIAGo* is going to throw the drink's can. The position of the bin is determined by the tree of transformations. It can be modified by changing the *parameters.launch* file.

### 5.1.2 Connection to *TIAGo*

In order to connect to *TIAGo* it is necessary to turn in on and execute the following commands at each terminal window:

---

```
1 source /ros/path/setup.bash
2 export ROS_MASTER_URI=http://tiagoioc:11311
3 export ROS_IP= ip of computer used
4 cd working_space
5 source devel/setup.bash
```

---

As an example, these are the commands that are needed to be executed in *aquaris* computer in *IOC* lab.

---

```
1 source /srv/robotica/ros/install_isolated/setup.bash
2 export ROS_MASTER_URI=http://tiagoioc:11311
3 export ROS_IP=147.83.37.19
4 cd catkin_ws
5 source devel/setup.bash
```

---

It is necessary to execute these commands to successfully connect the *ROS* master service running in the robot with the working environment on the computer.

### 5.1.3 Map building

In addition, it is necessary to build the environment map where *TIAGo* is going to navigate. If this map is not built, it is not possible for *TIAGo* to know where is him.

On the other hand, in order to have good transformations of the important points, and improve the *TIAGo* 's location, it is important to have two *ArUcos* at each table to locate better where is the robot in reference to the tables.

In order to build the map it is necessary to execute following steps:

1. Call *Rviz* and start mapping process executing the following commands:

---

```
1 roslaunch hackaton_nav rviz.launch
2 rosservice call /pal_navigation_sm "input: 'MAP'"
```

---

2. Move the robot around till you like the map:

---

```
1   rosservice call /pal_navigation_sm "input: 'LOC'"
```

---

\* If it doesn't work probably is because the " and ' have changed format in the terminal rewrite them.

3. Launch the following command:

---

```
1   roslaunch hackaton_nav hackaton_master.launch
```

---

4. Place the robot in a way that it detects both *ArUco* frames in sensing table and then run the command:

---

```
1   rosrun hackaton_nav connect_picking
```

---

5. Check in *Rviz* that the positions spawned around both tables are correct as described in the memory.

6. Place the robot in a way that it detects both *ArUco* frames in serving table and then run the command:

---

```
1   rosrun hackaton_nav connect_serving
```

---

7. Finally, save the aruco positions. This way it is possible to close and relaunch *hackaton\_master.launch* without losing *ArUco* references. To do that it is necessary to run the following command:

---

```
1   roslaunch hackaton_nav save_aruco_points.launch
```

---

## 5.2 How to reproduce the project

After setting up all the previous requirements, it will be possible to launch the entire system program.

### Launch program

In order to launch the program it is necessary to execute the following commands in different windows. Remember to execute the commands shown in section 5.1.2 in order to connect to TIAGO.

- roslaunch dmp dmp.launch
- rosrun dmp\_package dmp\_service.py
- roslaunch tiago\_manager task\_manager.launch
- rqt\_image\_view for image interface

## Program procedure

The program is going to execute the following steps. The bold steps are the ones where the user must interact.

1. Ask user for a drink.
  - (a) Image interface: at the beginning of the program a default image is shown in *rqt\_image\_view* window published in topic *tiago\_sensing/image*.
  - (b) TIAGo asks to the user if he/she wants a drink.
  - (c) **The user must say yes or no using the terminal where the *tiago\_manager task\_manager.launch* was launched.**
2. TIAGo goes to sensing table.
  - (a) A base movement is performed.
  - (b) TIAGo is prepared to detect cans.
3. Can choice.
  - (a) TIAGo detects drink cans.
  - (b) Show the resulting image after applying an image processing. In this image all the cans are labeled with a number.
  - (c) **The user must select, using the terminal, the number that indicates the drink that he/she wants.**
4. TIAGo goes to serving table.
  - (a) TIAGo grasps the chosen drink.
  - (b) TIAGo moves to the serving table.
  - (c) TIAGo serves the drink in a glass in serving table.
  - (d) Throws the can to a recycle bin next to the serving table.
  - (e) **User enjoys the drink.**

## 6 Costs

The economic cost of the project has been structured into three main parts. The first one is the depreciation related with the project of the robotics equipment and computers of the laboratory. The second one are the costs of the working hours of the students involved in the project as well as the supervisors that were consulted during its completion. Finally, an estimation of the electrical energy consumed during the tests and the development of the solution is also included.

The robotics equipment used to develop the projects consist only of *TIAGo* as none other robots or extra sensors or equipment was necessary. A robot of this characteristics has usually an estimated useful life of 8 years. Considering a use of *TIAGo* of 6 hours a day, 5 days a week, 48 weeks a year, this generates a useful life of the robot of 11520 hours. The two computers of the lab used to develop the project have, on the other hand, an estimated useful life of 5 years. This means that using them an average time of 10 hours a day gives an estimated useful life of 1200 hours. Depreciation can be then calculated from the total price of the robot and the computers, the knowledge of their useful life and the total time that they have been used.

The project has been completed from February to June with an average of 4 hours per day. This generates an approximate total time of 400 hours invested in development and testing. The two computers of the lab have been running constantly for almost all of these time so a percentage of 95% will be estimated for them, which means a total of 380 hours of use. *TIAGo* on the other hand has been working less and mainly in the final part of the project when the different tests have been performed, therefore a ratio of 60% will be considered for its working hours. This generates a total of 240 hours using the robot for this project.

The working hours of both students can be estimated of 400 hours employed in the lab to complete project plus 20 hours more spent in writing this report and completing other documentation tasks. This is a total of 420 hours spent by each student. As *ETSEIB* recommends, the salary for the students will be considered of 8 €/h. Supervision and meeting hours with the director of the project and other members of the staff of the laboratory will be considered of 50 hours in total with an average cost of 30 €/h.

Finally, the electrical cost can be summarized by only considering the energy consumed by the computers and the robot during their working hours. The rest of electrical elements of the room are not considered as there is much more people working on the lab and light consumption would remain the same regardless of the project analyzed in this memory. With an average electrical cost in 2019 of 0.15 €/kWh, the consumption of both computers can be estimated to be of about 0.40 kWh. *TIAGo*, on the other hand, has an electrical consumption of 0.72 kWh.

Next table presents all the costs described and analyzed. The total cost of the project finally is of 9964.75 €.

<b>Cost factor</b>	<b>Fixed cost [€]</b>	<b>Life expectancy [h]</b>	<b>Variable cost [€/h]</b>	<b>Time referred to the project [h]</b>	<b>Cost related to the project [€]</b>	
<b>TIAGo</b>	Depreciation	50000	11520	4.34	380	1649.31
	Electric consumption	-	-	0,11	380	41.04
<b>Lab Computers</b>	Depreciation	2000	12000	0.17	240	40
	Electric consumption	-	-	0,06	240	14.4
<b>Students</b>	-	-	8	840	6720	
<b>Supervisors</b>	-	-	30	50	1500	
<b>TOTAL COST</b>					<b>9964.75</b>	

Table 3: Calculation of the final cost of the project. (Variable costs of *TIAGo* and lab computers have been computed dividing their fixed cost by their life expectancy in hours. Variable costs of electric consumption of each systems have been computed multiplying the power consumption of each machine by the average price of electricity described before).

## 7 Environmental and social impact

When developing new ideas and solutions in all the fields of engineering it is always very interesting to take some time and reflect on the impact of that idea in the environment and in society. In this chapter a small discussion around these topics will be conducted.

### 7.1 Environmental impact

Robotics research can have a very deep impact in ecology and energy industries. In the particular field of this project, service robotics, the impact is much lower. In fact, using this kind of mobile manipulators in many different business could incur in an increment on energy consumption by this establishments. Of course this energy could be gathered using clean sources making the whole process clean and reducing the environmental impact of these solutions.

But by just analyzing the project and the solutions that might come from it and be developed around it, it doesn't seem very relevant to think that it might have any environmental impact. The relevance of this project from the environmental impact point of view is almost null.

### 7.2 Social impact

On the other hand, this kind of service robots will have a very huge impact at a socioeconomic level. The perfecting of these machines and their implantation on the service sector will have a huge impact in the working market and society in general. This kind of robots will be working in professional area likes, medicine, professional cleaning, construction and many a others but they will also reach our homes with domestic uses like vacuum cleaning, entertaining, etc.

For this reasons mobile manipulators like *TIAGo* will have a deep social impact in the short term future. First of all they will change our daily basis activities freeing up time and changing how we structure our daytime. If daily tasks like cleaning, preparing our food and other common activities are performed by robots instead of us, the impact in our way of life can be very deep. For this reason it will be important to change some cultural prejudices to integrate robots in our home life in a comfortable way that does not derive into mental and social problems.

On the other hand, the most important impact will occur in the job market. The massification of this kind of robots can be very dramatic if it is not performed in a controlled way. With robots completing tasks that are nowadays performed by humans, many jobs that are currently very common will disappear. In fact manual and service jobs that are associated with low preparation workers but also other important and very complex professions like surgeons, will change drastically and might become completely obsolete. On the other side, the appearance of all these new robots will create a new set of job position that will require different abilities and therefore education must change before the next generations are not prepared enough for the changes coming.

In summary, this project might be a very tiny step for the development of service robotics, and thus, it might contribute to all these change process making it an important thing to reflect about all these possible futures and which role must robotic engineers play in it.

## 8 Conclusions

This section is about the reached conclusions from perception, task manager and the entire process.

### 8.1 Specific conclusions

#### 8.1.1 Perception

As the initial pose estimation algorithm did not work quite well the optimization processes shown in section 2.3.2 has been required. After applying these processes the pose estimation of each of the cans improved a lot, but depending of the situation the pose accuracy continued not being quite good.

In order to accept the perception as good enough has to meet an important requirement: the maximum position error must be the slack between the two gripper fingers in order to make the can fit in between them and so grasp the can properly. Quantitatively, the maximum position error is  $\pm 0.75cm$ . This is computed taking into account the slack of gripper fingers and the size of the type of can used in this project.

Therefore, depending of the circumstances the pose estimation is good enough. These circumstances depend of three main factors:

- **The inefficient detection of the top of the can ellipse:** it would be necessary to improve the robustness of the algorithm to avoid problems with brightness. This problem is caused for the brightness environment and the characteristics of the cans, as their color.
- **The distance to the item:** The higher is the distance to the can that is wanted to be detected, the higher are the possibilities of detecting an ellipse that does not fit quite well with the top of the can obtaining a wrong position, or the more likely is to find a bad location of the center of the top of the can and the center of the hole obtaining a wrong orientation of the can.
- **The quality of the camera:** With a higher camera quality, the location accuracy in pixels would be higher and the distance would not affect too much to the position error.

On the other hand, the accuracy compared with the arucos's position accuracy is very similar depending of the situation. If the cans are in a good circumstance, according to the factors explained above (except the camera quality), their position is very similar from the one obtained detecting the aruco markers, sometimes even better. Otherwise, their position is worse than the given by the markers.

In conclusion, in certain situations the perception module is good enough to achieve its goal, but its robustness should improve to give it more reliability in any environment and location. In addition, a good advantage is that it can be adjusted to detect other can sizes and it is very easy to use. It is only necessary call a service which returns the estimated position of the detected objects.

### 8.1.2 Task manager

As it has been seen in this project, task manager was built with a behavior tree model using a visual editor called Groot. The use of this tool made much easier the definition of the system procedure and its scalability. Unlike hard coding, the task manager procedure to add, delete or modify functionalities is very easy. Moreover, it allows to follow the steps of the system execution much easier.

Concluding, it can be seen that using behavior trees is a good way to scale a program without going deep in programming, just creating nodes and building system execution strategy visually using Groot. In addition, it allows to separate each behavior tree node in order to have a better debugging process when needed, being possible to check each part of the process without many problems.

## 8.2 General conclusions

Before starting to analyze the whole waiter process it would be interesting to mention that the robotic platform suffered an accident that kept it out of work for almost two weeks in the last month of the project. This was a very important complication as most of the experimental and testing tasks were planned to be performed on this last month. Due to this, the time for testing and experimenting was reduced by a half so some of the tests had to be canceled and others could not be performed as desired. This also generated that some solutions designed after performing some of the tests could not be retested again before finishing this memory.

All the individual parts have been proven to work successfully and the task manager can correctly connect them in order to perform the whole waiter process. The solution developed can complete the entire serving process while sensing, picking and serving the cans as demanded by the users. The results obtained show that in simple scenarios, the method works correctly and allows the robot to be fully autonomous to serve drinks to the users. The task manager block, which gets its most importance in the final tests, has proven to correctly combine the different blocks and generate a sequence of actions to complete the task. The objective of completing the hackathon challenge by integrating the solutions completed for it and developing new ones has been achieved.

The solution has been structured in different packages that can be launched and tested separately, presenting independent tools to solve each one of the tasks included in each block. Aside from this, the implementation in a behavioural tree has provided a modular structure to modify the process easily using a graphical GUI. For all these reasons it can be affirmed that the objective of generating a modular solution that can be afterwards reused for other projects has been achieved too.

Despite these positive achievements, the method is not as robust as expected and still presents some critical flaws. The system is very sensitive to errors in perception and navigation that make the picking and serving trajectory generation algorithms fail in finding suitable plans to perform the movements. These errors get magnified when the inverse kinematics computation functions use this erroneous positions to compute the final configurations for grasping and serving. The arm manipulation package is the more sensitive to

these errors and also the most critical to assure a correct completion of the waiter process.

On the other hand, the computation of the inverse kinematics itself is also a problem as seen in the conclusions for arm manipulations. As the number of solutions provided by the library is limited and depends on the value of the redundant joints the configurations obtained to grasp a can might be very far away from the one obtained to serve it afterwards generating a lot of problems in the trajectories that lead to an execution failure. This mismatch between picking and serving configurations also diminish the robustness of the method as they reduce the number of scenarios in which these problems will not appear.

The complexity of the scene presented to the robot is still too relevant to assure a successful completion of the process. Although the sensing capabilities are ready to perceive picking tables very crowded with cans, the path planning capabilities are very limited and the cans placed in very complicated regions as well as the presence of too many cans can lead to the impossibility of finding a path to pick the desired can which leads to failing in completing the process.

Summing up, the whole methods has been proven to be suitable to perform the waiter task assigned and all the blocks and methods generated have been integrated and combined successfully to generate a proper solution. The problem is the robustness of the methods as all the errors previously detailed can make the whole process fail. The next step should be to enhance the solution in order to make it more robust against all these errors and uncertainties but the base method generated is proven to be a good foundation to keep on developing the solution.



## 9 Future work

After finishing this project, it has been observed that each of the modules that compose the entire system could be improved in order to have better results and a more robust system. Down below, the different possible improvements that could be applied to the module are presented for future enhancements.

### 9.1 Perception

The current perception module is very susceptible to having malfunction on pose estimation depending on the scene brightness and the distance of the cans from the camera. Therefore, as future work, in order to improve its robustness, it would be very useful to improve some important points:

- Improve image processing algorithm in order to avoid problems in front of brightness noise. It could be done by adding some techniques that support.
- The distance problem could be improved by adding other techniques that support the current one, even using some depth information.
- Improving camera quality or adding a new one, would increase the chances to find the correct coplanar points.
- Changing the type of cans, for ones with smaller size, would allow a higher error, increasing the probability of fitting between the tool fingers.

### 9.2 Task manager

The task manager could follow a more intuitive behavior tree. But a good feature to add is to allow the user to follow the nodes that are being executed at each time, by checking it with in a graphical way, using Groot.

On the other hand, it would be good to modify the behavior tree in a more robust way in order to apply contingency plans, minimizing the risk that the whole system process does not work correctly.

### 9.3 General process

The future work should focus on diminishing the errors that make the system unreliable and on trying to find the best way to increase its robustness. The best idea would be to try to generate functions and methods that check the viability of the positions detected by the perception tools and if they are too erroneous find a support strategy to correct them.



## 10 Acknowledgements

I would like to express my thanks to our advisor Jan Rosell for the opportunity of working on this project and providing us with his help and patience whenever it was needed.

On the other hand, I would like to thank people from robotics laboratory of IOC, especially Leopold Palomo, for his help with all the laboratory equipment used and the technical problems we found during the project development and Nestor García Hidalgo for helping us with his knowledge related to TIAGo, robotics and some previous implementations in TIAGo.

Finally, I want to express my thanks to the company of my colleague Xavier Garcia Peroy during the development of this project. There were some complicated moments during the development that with his help and patience were easier to overcome. I appreciate the good work he has done hard and effectively.



## References

- [1] IROS 2018 Mobile Manipulation Hackathon web page, "IROS 2018 MMH Gallery," Last access: 21/06/2019. [Online]. Available: [http://iros18-mmh.pal-robotics.com/#iLightbox\[gallery\\_image\\_1\]/21](http://iros18-mmh.pal-robotics.com/#iLightbox[gallery_image_1]/21)
- [2] Internation Federation of Robotics, "Service robotics," Last access: 23/05/2019. [Online]. Available: <https://ifr.org/service-robots>
- [3] Open Source Robotics Foundation, "Robot Operating System (ROS)," <https://www.ros.org>, [Online; accessed 24/06/2019].
- [4] P. Robotics, "TIAGo, The mobile manipulator," Last access: 23/05/2019. [Online]. Available: <http://tiago.pal-robotics.com/>
- [5] C. L. Kinkade, "Meet Ginger, the robot waiter," Last access: 23/05/2019. [Online]. Available: <https://edition.cnn.com/videos/world/2018/12/03/robot-waiter-nepal-kinkade-dnt-cnni-vpx.cnn>
- [6] Vocativ, "Robots Are The Waiters At This Restaurant," Last access: 23/05/2019. [Online]. Available: <https://www.youtube.com/watch?v=kvzrP2lrvqA>
- [7] P. Robotics, "IROS 2018 mobile manipulation hackathon," Last access: 23/05/2019. [Online]. Available: <http://iros18-mmh.pal-robotics.com/>
- [8] I. Corporation, W. Garage, and Itseez, "OpenCV," Last access: 20/06/2019. [Online]. Available: <https://opencv.org/>
- [9] E. Riba, "Real Time pose estimation of a textured object," Last access: 21/06/2019. [Online]. Available: [https://docs.opencv.org/master/dc/d2c/tutorial\\_real\\_time\\_pose.html](https://docs.opencv.org/master/dc/d2c/tutorial_real_time_pose.html)
- [10] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling Planning and Control*. Springer Publishing Company, Incorporated, 2008, ch. Visual Servoing.
- [11] A. Burri, "Tinyfsm," Last access: 16/06/2019. [Online]. Available: <https://github.com/digint/tinyfsm/tree/master/examples/elevator>
- [12] A. Neumann, "Transitions," Last access: 16/06/2019. [Online]. Available: <https://github.com/pytransitions/transitions>
- [13] M. Colledanchise, "Github repository: BehaviorTree," Last access: 23/05/2019. [Online]. Available: <https://github.com/BehaviorTree/BehaviorTree.CPP>
- [14] —, "Github repository: Groot," Last access: 23/05/2019. [Online]. Available: <https://github.com/BehaviorTree/Groot>
- [15] "NLopt Reference," Last access: 09/06/2019. [Online]. Available: [https://nlopt.readthedocs.io/en/latest/NLopt\\_Reference/](https://nlopt.readthedocs.io/en/latest/NLopt_Reference/)

- [16] S. G. Johnson, "NLopt git repository," Last access: 09/06/2019. [Online]. Available: <https://github.com/stevengj/nlopt>
- [17] M. Colledanchise, "BehaviorTree basics," Last access: 23/05/2019. [Online]. Available: [https://www.behaviortree.dev/bt\\_basics/](https://www.behaviortree.dev/bt_basics/)
- [18] ——, "BehaviorTree XML format," Last access: 16/06/2019. [Online]. Available: [https://www.behaviortree.dev/xml\\_format/](https://www.behaviortree.dev/xml_format/)

## A Code Repository

All the software packages developed have been uploaded into the *IOC* robotics lab repository. The link is presented next.

[https://gitioc.upc.edu/xavier.garcia/TIAGO\\_waiter](https://gitioc.upc.edu/xavier.garcia/TIAGO_waiter)

The repository includes a *README* file explaining the structure of all the packages and a summary of the user guide in order to run the methods developed.

A link to a *Wiki* page with all the videos of the experiments is included next as mentioned in the section [4](#).

[https://gitioc.upc.edu/xavier.garcia/TIAGO\\_waiter/wikis/home](https://gitioc.upc.edu/xavier.garcia/TIAGO_waiter/wikis/home)



## B XML file of the project

Listing 8: XML file that describes the behavior of the Behavior Tree nodes that defines the operation of this project.

```

1   <root main_tree_to_execute="BehaviorTree">
2   <!-->
3   <BehaviorTree ID="ASK USER TO CHOOSE DRINK">
4
5       <Sequence>
6           <RetryUntilSuccessful num_attempts="1">
7               <Sequence>
8                   <Action ID="detectCans"/>
9                   <Condition ID="isThereSomeCan"/>
10                  </Sequence>
11             </RetryUntilSuccessful>
12             <Action ID="askUserForCanChoice"/>
13             <Action ID="confirmCanSelected"/>
14         </Sequence>
15
16     </BehaviorTree>
17     <!-->
18     <BehaviorTree ID="BehaviorTree">
19
20         <RetryUntilSuccessful num_attempts="1">
21             <Inverter>
22                 <Sequence>
23                     <SubTree ID="OPEN BAR"/>
24                     <SubTree ID="SERVING TABLE USER INTERACTION"/>
25                     <SubTree ID="GO TO SENSING TABLE"/>
26                     <SubTree ID="ASK USER TO CHOOSE DRINK"/>
27                     <SubTree ID="GRASP CAN"/>
28                     <SubTree ID="GO TO SERVING TABLE"/>
29                     <SubTree ID="SERVE DRINK"/>
30                 </Sequence>
31             </Inverter>
32         </RetryUntilSuccessful>
33
34     </BehaviorTree>
35     <!-->
36     <BehaviorTree ID="GO TO SENSING TABLE">
37
38         <Sequence>
39             <RetryUntilSuccessful num_attempts="1">
40                 <Fallback>
41                     <Sequence>
42                         <Action ID="prepareToGoToSensingTable"/>

```

```
43             <RetryUntilSuccessful num_attempts="1">
44                 <Sequence>
45                     <Action ID="goToSensingTable"/>
46                     <Condition ID=
47                         isRobotLocalizedSensingTable"/>
48                 </Sequence>
49             </RetryUntilSuccessful>
50         </Sequence>
51         <Inverter>
52             <Condition ID="isGoToSensingTableSolved"/>
53         </Inverter>
54     </Fallback>
55     </RetryUntilSuccessful>
56     <Condition ID="isRobotLocalizedSensingTable"/>
57 </Sequence>
58
59 </BehaviorTree>
60 <!-->
61 <BehaviorTree ID="GO TO SERVING TABLE">
62
63     <Sequence>
64         <RetryUntilSuccessful num_attempts="5">
65             <Fallback>
66                 <Sequence>
67                     <Action ID="prepareToGoToServingTable"/>
68                     <RetryUntilSuccessful num_attempts="1">
69                         <Sequence>
70                             <Action ID="goToServingTableEnd"/>
71                             <Condition ID=
72                                 isRobotLocalizedServingTable"/>
73                         </Sequence>
74                     </RetryUntilSuccessful>
75                 </Sequence>
76                 <Inverter>
77                     <Condition ID="isGoToServingTableSolved"/>
78                 </Inverter>
79             </Fallback>
80             </RetryUntilSuccessful>
81             <Condition ID="isRobotLocalizedServingTable"/>
82         </Sequence>
83
84     </BehaviorTree>
85 <!-->
86     <BehaviorTree ID="GRASP CAN">
87
88         <RetryUntilSuccessful num_attempts="1">
```

```

87     <Sequence>
88         <Action ID="prepareToGraspCan"/>
89         <RetryUntilSuccessful num_attempts="1">
90             <Fallback>
91                 <Action ID="computeInverseKinematics"/>
92                 <Inverter>
93                     <Action ID="correctPositionIK"/>
94                 </Inverter>
95             </Fallback>
96         </RetryUntilSuccessful>
97         <Action ID="prepareToComputePath"/>
98         <RetryUntilSuccessful num_attempts="1">
99             <Fallback>
100                 <Action ID="computePath"/>
101                 <Inverter>
102                     <Action ID="correctPositionPath"/>
103                 </Inverter>
104             </Fallback>
105         </RetryUntilSuccessful>
106         <Action ID="tryToGraspCan"/>
107     </Sequence>
108 </RetryUntilSuccessful>
109
110 </BehaviorTree>
111 <!-->
112 <BehaviorTree ID="OPEN BAR">
113
114     <Sequence>
115         <Action ID="openBar"/>
116     </Sequence>
117
118 </BehaviorTree>
119 <!-->
120 <BehaviorTree ID="SERVE DRINK">
121
122     <Sequence>
123         <Action ID="serveDrink"/>
124         <Action ID="closeBar"/>
125     </Sequence>
126
127 </BehaviorTree>
128 <!-->
129 <BehaviorTree ID="SERVING TABLE USER INTERACTION">
130
131     <Sequence>
132         <Action ID="goToServingTable"/>

```

```
133             <Action ID="askForDrinks"/>
134         </Sequence>
135
136     </BehaviorTree>
137     <!--
138     <BehaviorTree ID="SETUP">
139
140         <Sequence>
141             <Action ID="initNodes"/>
142             <Action ID="readParameters"/>
143             <Action ID="transformStamped"/>
144             <Action ID="headTowardUser"/>
145         </Sequence>
146
147     </BehaviorTree>
148     <!--
149     <TreeNodesModel>
150         <SubTree ID="ASK USER TO CHOOSE DRINK"/>
151         <SubTree ID="GO TO SENSING TABLE"/>
152         <SubTree ID="GO TO SERVING TABLE"/>
153         <SubTree ID="GRASP CAN"/>
154         <SubTree ID="OPEN BAR"/>
155         <SubTree ID="SERVE DRINK"/>
156         <SubTree ID="SERVING TABLE USER INTERACTION"/>
157         <SubTree ID="SETUP"/>
158         <Action ID="askForDrinks"/>
159         <Action ID="askUserForCanChoice"/>
160         <Action ID="askUserToSolveProblem"/>
161         <Action ID="closeBar"/>
162         <Action ID="computeInverseKinematics"/>
163         <Action ID="computePath"/>
164         <Action ID="confirmCanSelected"/>
165         <Action ID="correctPositionIK"/>
166         <Action ID="correctPositionPath"/>
167         <Action ID="detectCans"/>
168         <Action ID="goToSensingTable"/>
169         <Action ID="goToSensingTableFailure"/>
170         <Action ID="goToServingTable"/>
171         <Action ID="goToServingTableEnd"/>
172         <Action ID="headTowardUser"/>
173         <Action ID="initNodes"/>
174         <Action ID="initializeVariables"/>
175         <Condition ID="isCanGrasped"/>
176         <Condition ID="isGoToSensingTableSolved"/>
177         <Condition ID="isGoToServingTableSolved"/>
178         <Condition ID="isPreviousDrinkServedCorrecly"/>
```

```
179      <Condition ID="isRobotLocalizedSensingTable"/>
180      <Condition ID="isRobotLocalizedServingTable"/>
181      <Condition ID="isThereSomeCan"/>
182      <Condition ID="isValidPath"/>
183      <Action ID="openBar"/>
184      <Action ID="prepareToComputePath"/>
185      <Action ID="prepareToGoToSensingTable"/>
186      <Action ID="prepareToGoToServingTable"/>
187      <Action ID="prepareToGraspCan"/>
188      <Action ID="readParameters"/>
189      <Action ID="serveDrink"/>
190      <Action ID="transformStamped"/>
191      <Action ID="tryToGraspCan"/>
192  </TreeNodesModel>
193  <!------->
194 </root>
```



## C Pose estimation algorithm

In this appendix section it will be explained the steps to follow in order to get the position of the detected objects with the use of the perception module. All the calculus below has been extracted from *Modelling, Planning and Control* book [10] and are implemented inside *perception* module in *pose\_estimation.cpp* file in a function called *CanDetector::poseEstimation*.

### C.1 The feature vector

Let  $\{B\}$  be the world reference frame and  $\mathbf{T}_c^b$  be the transformation that defines the reference frame of a camera w.r.t.  $\{B\}$ . If  $\mathbf{P} = [p_x, p_y, p_z]^T$  is a point and  $\tilde{\mathbf{p}} = [\mathbf{p}^T, 1]^T$  its generalized representation, then the coordinates from the reference frame of the camera are:

$$\tilde{\mathbf{p}}^c = [p_x^c, p_y^c, p_z^c, 1]^T = \mathbf{T}_c^b \tilde{\mathbf{p}} \quad (9)$$

The perspective transformation of a camera with distance focus  $f$  leads to the following values (Fig. 42):

$$\begin{aligned} \mathbf{X}_f &= -f \frac{p_x^c}{p_z^c} \\ \mathbf{Y}_f &= -f \frac{p_y^c}{p_z^c}, \end{aligned} \quad (10)$$

And considering a "virtual" image plane located at  $z_c = f$  and defining  $\mathbf{X} = \frac{p_x^c}{p_z^c}$  and  $\mathbf{Y} = \frac{p_y^c}{p_z^c}$ , the expressions in Eq. (10) become:

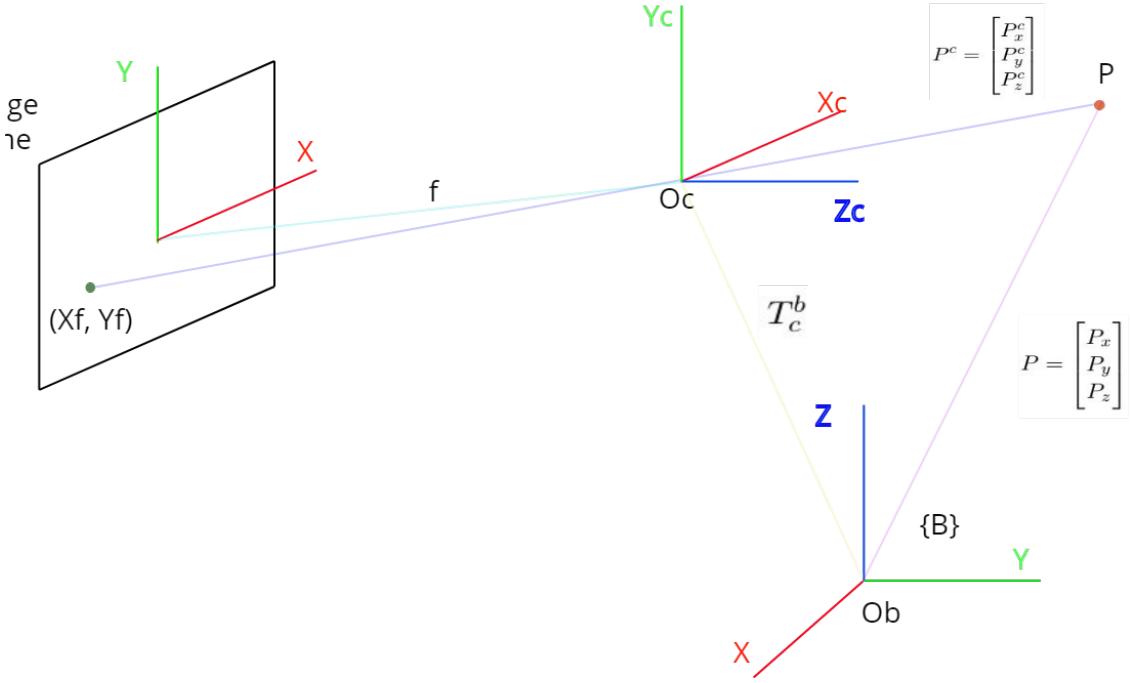
$$\begin{aligned} \mathbf{X}_f &= f \mathbf{X} \\ \mathbf{Y}_f &= f \mathbf{Y} \end{aligned} \quad (11)$$

The representation in the virtual image plane of point  $[\mathbf{X}_f, \mathbf{Y}_f]$  depends on the spatial sampling (the spatial sampling units are the pixels). Pixel coordinates are obtained through a scaling process:

$$\begin{aligned} \mathbf{X}_I &= f \alpha_x \mathbf{X} + \mathbf{X}_0 \\ \mathbf{Y}_I &= f \alpha_y \mathbf{Y} + \mathbf{Y}_0, \end{aligned} \quad (12)$$

with  $\mathbf{X}_0$ , and  $\mathbf{Y}_0$ , being the offsets of the origin in the pixel coordinate system. This can be expressed in matrix form as:

$$\begin{bmatrix} \mathbf{X}_I \\ \mathbf{Y}_I \\ 1 \end{bmatrix} = \begin{bmatrix} f \alpha_x & 0 & \mathbf{X}_0 \\ 0 & f \alpha_y & \mathbf{Y}_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \\ 1 \end{bmatrix} = \Omega \begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \\ 1 \end{bmatrix} \quad (13)$$

Figure 42: Observation of a point  $P$ .

The parameters  $(\alpha_x, \alpha_y, \mathbf{X}_0, \mathbf{Y}_0, f)$  are the intrinsic parameters of the camera, and are assumed to be known. Then, we will work with the coordinates  $[\mathbf{X}, \mathbf{Y}]^T$  which will be called *normalized coordinates*. The relationship between these coordinates and the 3D coordinates of the point w.r.t. the camera frame can be expressed as:

$$\lambda \begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \\ 1 \end{bmatrix} = \Pi \begin{bmatrix} p_x^c \\ p_y^c \\ p_z^c \\ 1 \end{bmatrix}, \quad (14)$$

with:

$$\Pi = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \quad (15)$$

Then, the **feature vector of a point** will be defined as:

$$\mathbf{s} = [\mathbf{X}, \mathbf{Y}]^T, \quad (16)$$

and its expression in generalized coordinates as:

$$\tilde{\mathbf{s}} = [\mathbf{X}, \mathbf{Y}, 1]^T. \quad (17)$$

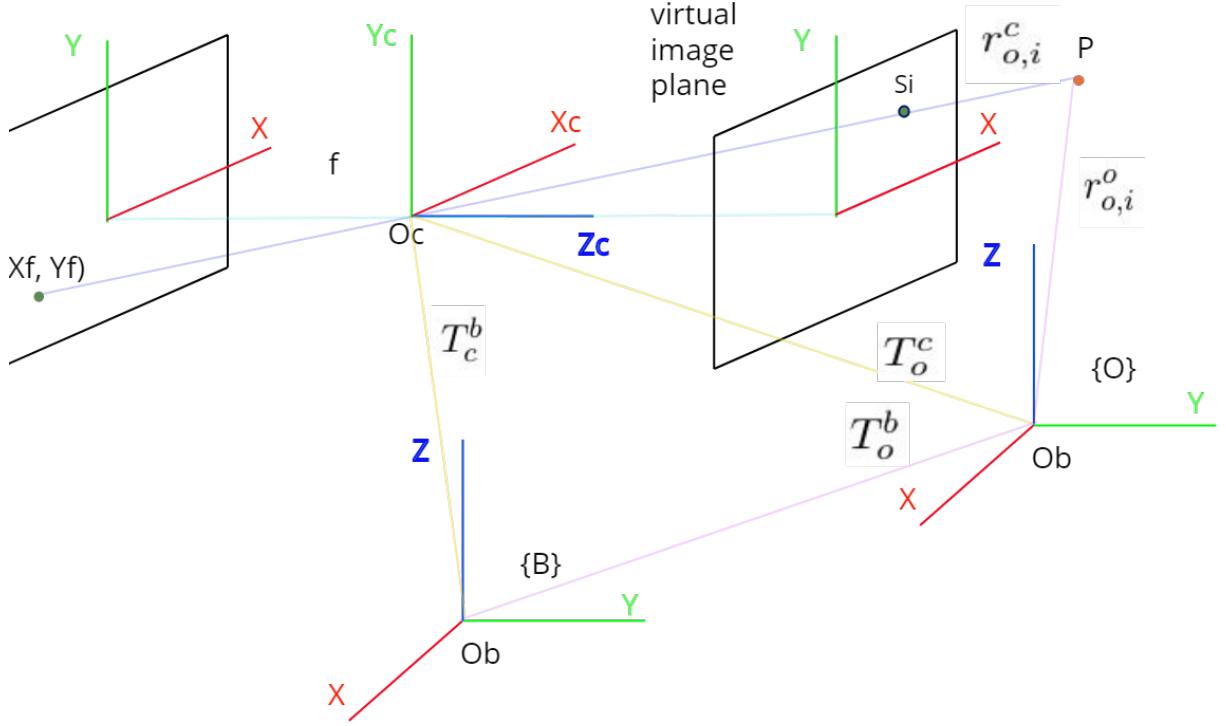


Figure 43: Problem setup.

The expression relating the feature vector and the corresponding point in the object is:

$$\lambda \tilde{s} = \Pi \tilde{p}^c \quad (18)$$

The feature vector can be extended to a set of points:

$$\mathbf{s} = \begin{bmatrix} \mathbf{s}_1 \\ \vdots \\ \mathbf{s}_n \end{bmatrix}, \text{ with } \mathbf{s}_i = \begin{bmatrix} \mathbf{X}_i \\ \mathbf{Y}_i \end{bmatrix}. \quad (19)$$

## C.2 Analytic solution

Given a set of points in the object and the associated feature vector, find the pose of the object w.r.t. the camera, i.e.  $\mathbf{T}_o^c$  (Fig. 43).

Consider the point  $P_i$  on the object. Its coordinates in the camera frame are:

$$\tilde{\mathbf{r}}_{o,i}^c = \mathbf{T}_o^c \tilde{\mathbf{r}}_{o,i}^o \quad (20)$$

And the corresponding feature vector in the image plane is  $\mathbf{s}_i = [\mathbf{X}_i, \mathbf{Y}_i]^T$ , that is related with  $\tilde{\mathbf{r}}_{o,i}^o$  with the expression:

$$\lambda_i \tilde{\mathbf{s}}_i = \Pi \mathbf{T}_o^c \tilde{\mathbf{r}}_{o,i}^o \quad (21)$$

Considering  $n$  points, a systems of equations results:

$$\begin{aligned}\lambda_1 \tilde{\mathbf{s}}_1 &= \Pi \mathbf{T}_o^c \tilde{\mathbf{r}}_{o,1}^o \\ &\vdots \\ \lambda_n \tilde{\mathbf{s}}_n &= \Pi \mathbf{T}_o^c \tilde{\mathbf{r}}_{o,n}^o,\end{aligned}\quad (22)$$

This system can be solved for the unknown elements of  $\mathbf{T}_o^c$ . This is a difficult issue with possibly multiple solutions, that depends on the number of points and on whether they are coplanar or not. In the case of coplanar points it is affordable with the following procedure:

1. Let  $\mathbf{S}(\cdot)$  represent the skew-symmetric matrix:

$$\mathbf{S}([x, y, z]^T) = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \quad (23)$$

Apply it to the feature vector:

$$\mathbf{S}(\tilde{\mathbf{s}}_i) = \begin{bmatrix} 0 & -1 & \mathbf{Y}_i \\ 1 & 0 & -\mathbf{X}_i \\ -\mathbf{Y}_i & \mathbf{X}_i & 0 \end{bmatrix} \text{ with } \tilde{\mathbf{s}}_i = \begin{bmatrix} \mathbf{X}_i \\ \mathbf{Y}_i \\ 1 \end{bmatrix} \quad (24)$$

2. Multiply  $\mathbf{S}(\tilde{\mathbf{s}}_i)$  to both sides of  $\lambda_i \tilde{\mathbf{s}}_i = \Pi \mathbf{T}_o^c \tilde{\mathbf{r}}_{o,i}^o$ , i.e.:

$$\mathbf{S}(\tilde{\mathbf{s}}_i) \lambda_i \tilde{\mathbf{s}}_i = \mathbf{S}(\tilde{\mathbf{s}}_i) \Pi \mathbf{T}_o^c \tilde{\mathbf{r}}_{o,i}^o \quad (25)$$

The left part is the zero vector:

$$\begin{bmatrix} 0 & -1 & \mathbf{Y}_i \\ 1 & 0 & -\mathbf{X}_i \\ -\mathbf{Y}_i & \mathbf{X}_i & 0 \end{bmatrix} \begin{bmatrix} \lambda_i \mathbf{X}_i \\ \lambda_i \mathbf{Y}_i \\ \lambda_i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (26)$$

From the right part, let define  $\mathbf{H}$  as:

$$\begin{aligned}\mathbf{H} &= \Pi \mathbf{T}_o^c = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 & \mathbf{o}_{c,o}^c \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 & \mathbf{o}_{c,o}^c \end{bmatrix}\end{aligned}\quad (27)$$

3. Make the following simplification. Consider that all the points are in the plane  $z = 0$ , i.e.  $\mathbf{r}_{o,i}^o = [r_{x,i}, r_{y,i}, 0]^T$ . Then take only the non-null components of the points, redefine  $\mathbf{H}$  as  $\mathbf{H} = [\mathbf{r}_1 \ \mathbf{r}_2 \ \mathbf{o}_{c,o}^c]$ , and rewrite Eq. (25) as:

$$\mathbf{0} = \mathbf{S}(\tilde{\mathbf{s}}) \mathbf{H} [r_{x,i}, r_{y,i}, 1]^T \quad (28)$$

4. Rewrite Eq. (28) as:

$$\mathbf{A}_i(\mathbf{s}_i)\mathbf{h} = \mathbf{0} \quad (29)$$

with:

- $\mathbf{h}$  the  $9 \times 1$  vector obtained by stacking the components of  $\mathbf{H}$ , i.e.  $\mathbf{h} = [\mathbf{r}_1, \mathbf{r}_2, \mathbf{o}_{c,o}^c]^T$
- $\mathbf{A}_i(\mathbf{s}_i)$  a  $3 \times 9$  matrix:

$$\mathbf{A}(\mathbf{s}_i) = [r_{x,i}\mathbf{S}(\tilde{\mathbf{s}}_i) \ r_{y,i}\mathbf{S}(\tilde{\mathbf{s}}_i) \ \mathbf{S}(\tilde{\mathbf{s}}_i)] \quad (30)$$

The rank of  $\mathbf{A}(\mathbf{s}_i)$  is at most two, which is the rank of  $\mathbf{S}(\tilde{\mathbf{s}}_i)$ .

5. Consider 4 coplanar points, with no triplets of colinear points. Then the rank of:

$$\mathbf{A}(\mathbf{s}) = \begin{bmatrix} \mathbf{A}_1(\mathbf{s}_1) \\ \mathbf{A}_2(\mathbf{s}_2) \\ \mathbf{A}_3(\mathbf{s}_3) \\ \mathbf{A}_4(\mathbf{s}_4) \end{bmatrix}, \quad (31)$$

is eight and the system of equations  $\mathbf{A}(\mathbf{s})\mathbf{h} = \mathbf{0}$  can be solved up to a scaling factor<sup>2</sup>, i.e. if the solution is  $\xi\mathbf{h} = \xi[\mathbf{h}_1 \ \mathbf{h}_2 \ \mathbf{h}_3]^T$ , then:

$$\begin{aligned} \mathbf{r}_1 &= \xi\mathbf{h}_1 \\ \mathbf{r}_2 &= \xi\mathbf{h}_2 \\ \mathbf{o}_{c,o}^c &= \xi\mathbf{h}_3 \end{aligned} \quad (32)$$

and  $\xi$  is computed by imposing the unit norm constraint of vectors  $\mathbf{r}_1$  and  $\mathbf{r}_2$ :

$$|\xi| = \frac{1}{\|\mathbf{h}_1\|} = \frac{1}{\|\mathbf{h}_2\|} \quad (33)$$

The sign is chosen such that the solution places the object in front of the camera. Finally,  $\mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2$ .

---

<sup>2</sup>The null space of  $\mathbf{A}$  has dimension 1:  $\mathcal{N} = \xi\mathbf{v}$ .



## D Optimization of pose estimation

### D.1 Pose correction

This optimization process is in charge of correcting the position and orientation of the cans. In order to apply this correction, some steps are needed to be applied inside a function called *pose2D\_estimation* inside file *pose\_estimation.cpp* in *perception* module. This function receives the following parameters as input:

- $T_c^r$ : *xtion\_rgb\_optical\_frame* w.r.t *base\_footprint frame*
- A: 12x9 matrix obtained applying pose estimation algorithm. See Appendix C to know how.
- pose.position.x: obtained from pose estimation algorithm. It is the initial guess of can position in X axis.
- pose.position.y: obtained from pose estimation algorithm. It is the initial guess of can position in Y axis.
- alpha ( $\alpha$ ): angle formed by  $s_1$  and  $s_3$  points. Initial guess of the orientation of the can.
- $T_o^r$ : It is the result obtained of the position of the object w.r.t *base\_footprint frame* after applying pose correction.

Running this function, the following steps are performed:

- At each iteration of the optimization, the position of the object wr.r.t the robot is computed with the elements x, y and yaw, recomputed by the *nlopt* optimizer.
- Then, the position of the object is computed w.r.t. the camera of TIAGo.
- It is applied the following equation:

$$h = \begin{bmatrix} r_1 & r_2 & o_{c,o}^c \end{bmatrix}$$

where:

$$r_1 = T_o^c[:, 1]$$

$$r_2 = T_o^c[:, 2]$$

$$o_{c,o}^c = T_o^c[:, 4]$$

- Finally, the expression to optimize is described below. This expression must be minimized.

$$h^T A^T A h$$

## D.2 Orientation correction

This optimization process is in charge of correcting just the orientation of the cans, unlike the previous correction this is focused just in the orientation of the can. In order to do it, a series of steps are applied inside a function called *angle\_correction* inside file *pose\_estimation.cpp* in *perception* module. This function receives the following parameters as input:

- $T_c^r$ : *xtion\_rgb\_optical\_frame* w.r.t *base\_footprint frame*
- real center: center of the can computed by the pose estimation algorithm.
- pixelated center: position of the center of the can in image coordinates (pixels).
- $s_1$ : point  $s_1$  in image coordinates (pixels)
- cam: intrinsic camera parameters
- alpha ( $\alpha$ ): angle formed by  $s_1$  and  $s_3$  points taking into account the correction computed in [pose correction](#).

Now, the function follows the steps listed below to apply angle correction:

- A lower and an upper bounds are defined for yaw orientation.
- Necessary data to use the objective function is defined as *ObjectiveFunctionDataCorrection data*.
  - `data.cam = cam [intrinsic camera parameters]`
  - `data.T_camera_world = T_world_camera.inverse()`
  - `data.s1 = s1 [s1 point in image coordinates]`
  - `data.s_center = s_center [center of the can in image coordinates]`
  - `data.canCenter = centerCan [real position of the center of the can computed by the pose estimation algorithm]`
- Apply objective function:
  - Compute the center of the can in camera coordinates

$$P_{cc}^c = T_w^c * P_{cc}^w$$

- Compute  $S_1$  point in the camera coordinates.

$$yaw = \alpha \text{ [this is the value to optimize]}$$

$$S_1^w = \begin{bmatrix} P_{cc}^w \cdot x + r * \cos(yaw) \\ P_{cc}^w \cdot y + r * \sin(yaw) \\ P_{cc}^w \cdot z \\ 1 \end{bmatrix}$$

$$S_1^c = T_w^c * S_1^w$$

where  $\mathbf{r} = 0.026$  [m]. This is the distance in meters from the center of the can to the coplanar points. In this case is a circular object, so the distance from the center to these points is the radius, 0.026 [m] for a standard soda can.

- Compute projection of center of the can and  $S_1$  point in image plane, being  $s_1$  in image coordinates.

In order to compute this projection it is necessary to apply the equations (9), (10), (11), (12) and (13) from appendix C

- Correct  $s_1$  according to the difference between the center of the can computed in image coordinates computed by the image processing algorithm ( $c_{ip}$ ) and the center of the can that comes from the projection in image plane of the position estimated ( $c_{ps}$ ).

$$s_1c = s_1 + (c_{ip} - c_{ps})$$

- Compute  $v_s$  with the vector from center ( $c_{ip}$ ) to  $s_1$ .

$$v_s = s_1 - c_{ip}$$

- Compute  $v_p$  with the vector from  $s_1c$  to center ( $c_{ip}$ )

$$v_p = s_1c - c_{ip}$$

- Compute angle between vectors. This angle should be minimized.

$$\alpha = \text{acos} \left( \frac{\langle v_s, v_p \rangle}{\|v_s\| \|v_p\|} \right)$$