POLITECNICO DI TORINO

Master's Degree in Mechatronic engineering

Master's Degree Thesis

Design and implementation of a voice controlled pick and place robotic arm

Supervisors

Prof. Massimo VIOLANTE

Candidate

Andrea SPAMPINATO

2019-2020

Summary

The thesis is linked to the ALOHA research project [1] and covers the investigation, development and evaluation of an embedded keyword spotting system for industrial applications. The specific goal of the thesis is the implementation of the modules necessary for the integration of the speech module with a robotic arm (Comau e.Do), the addition of the perception functionalities and the realization of a demonstrator showing all the functionalities working together.

To do so, the thesis will be structured approximately as follow:

- Stereovision and feature extraction from pointcloud data;
- Documentation about the behaviour of the movegroup interface;
- Generation and filtering of grasps for sensed objects;
- Integration of pointcloud data and grasp generation into a demonstrator.
- Simulation of the demonstrator on Gazebo, to correctly calibrate both camera and robot before the real demo implementation.

The demo to be implemented will consider only static target objects placed on a support planar surface coming from a conveyor belt. The aim of the demo is to sense and find a way to manage some specific objects, accordingly with the input voice command coming from the KWS algorithm, running on the SensorTile board.

Basically, an operator can identify the target object as *correct* or *faulty* and the robotic arm, accordingly with the voice command received, must be able to distinguish and behave differently for these two cases. In principle, the demo will be configured to perform the following tasks:

- Sense and add relevant object to the planning scene. Perform some pointcloud preprocessing and identify objects of interest and their relative features;
- Find grasp poses. Possible grasps will be generated and filtered according to some specific criteria, in order to obtain a vector of feasible grasps for the object of interest;

- Manage the object of interest. Accordingly with the results coming from grasping, the robotic arm will be moved to pick up the target object if a feasible plan is found;
- Wait input voice command for placing. Here a voice command will be sensed by the KWS system running on SensorTile, that will recognize it with a certain precision and will send it as serial command to the demo node;
- Place the target object. Accordingly with the received voice command, the robotic arm will place the object of interest in a predefined target pose with a specified orientation;
- Go back to rest position and wait new inputs. Look for other objects of interest and wait for a new input.

Acknowledgements

Finally the long awaited day has come, writing these acknowledgments is the finishing touch to my thesis work. This period represents one of the most difficult challenge I have ever faced but it has been also a period of deep growth, not only from the scientific point of view, but also personally. It allows me to understand the complexity behind the development and realization of a real case application, within a completely new scenario, much more dynamic with respect to an academic one.

I would like to spend few words to all the people who supported and helped me during this period.

Firstly, I would like to thank all the colleagues met during this internship at Concept Reply. In particular, I turn to M. Di Florio and C. Chesta, that gave me the chance to work to this project, providing me with everything I needed to understand the project goals to reach.

A special thank goes to L. Rinelli, a kind of "mentor" for all the aspects linked to the technical development of the thesis work. I would not have been able to get the desired results without his help.

From the academic side, I would like to thank my supervisior M. Violante, that despite all the problem faced during the work, has always been by my side, recommending the best for the realization of a good work. A special thank goes also to S. Primatesta, that even having no interest in the development of this work, was always available to suggest me how to go on during the most trivial steps of this work.

Obviously a special thank goes to my family, that gave me the possibility to reach this important milestone, supporting me during the darkest periods of all this trip.

A special thank goes to my girlfriend C. Buccarelli, for helping me overcome seemingly insurmountable steps through simple but very effective gestures. Thanks for all the sleepless nights spent studying and for all the chats that gave me the strength to go on when I thought of giving up everything.

Last but not least, I would like to thank all my friends. We always supported each other, in moments of joy and sadness, spurring ourselves to reach the goal.

A heartfelt thanks to all!

"Andrea Spampinato"
Torino, 2019-2020

Table of Contents

List of Tables x

List of Figures xi

Acronyms xv

1 Deep Learning at the Edge using ALOHA Toolflow 1
- 1.1 Toolflow Description and Application . . . 2
- 1.2 Command Recognition in Smart Industry Applications . . . 5
  - 1.2.1 Baseline Dataset Preparation . . . 5
  - 1.2.2 Target Embedded Hardware . . . 7
  - 1.2.3 Baseline CNN Algorithm . . . 8
  - 1.2.4 Baseline Constraints . . . 11
- 1.3 System Design and Implementation . . . 13
- 1.4 References . . . 16

2 Robotic Arm Control 17
- 2.1 e.Do Cobot . . . 17
- 2.2 Mechanical Characteristics and Kinematics . . . 17
  - 2.2.1 Direct Kinematics . . . 17
  - 2.2.2 Inverse Kinematics . . . 23
- 2.3 Development Environments . . . 26
  - 2.3.1 Robot Operating System . . . 26
  - 2.3.2 Rviz . . . 27
  - 2.3.3 MoveIt . . . 27
- 2.4 Connection to the Robot . . . 28

3 Perception Pipeline 29
- 3.1 Intel RealSense D435i . . . 30
- 3.2 PCL Library . . . 32
  - 3.2.1 Filter in ROI . . . 33

3.2.2 Statistical Outlier Removal Filter . . . 33
3.2.3 Removal of Planar Surface . . . 34
3.2.4 Euclidean Clusters Extraction . . . 35
3.2.5 Update the Planning Scene . . . 47
3.3 Obtained Output . . . 48

4 Planning of Robot Movements 53
4.1 Pick Place Pipeline . . . 54
4.1.1 Generate the Planning Scene . . . 54
4.1.2 Pick . . . 56
4.1.3 Place . . . 60
4.2 Grasp Pipeline . . . 62
4.2.1 Robot-Agnostic Configuration . . . 66
4.2.2 Additional Configuration . . . 66

5 Demonstrator Design and Results 69
5.1 Setup Planning Scene . . . 70
5.2 Object Generation . . . 70
5.2.1 Random Generator . . . 71
5.2.2 Perception Generator . . . 73
5.3 Target Objects Management . . . 73
5.3.1 Grasps Computation . . . 76
5.3.2 Pick-Place . . . 77
5.4 Return to Idle State . . . 79
5.5 How to Use It . . . 80

6 Gazebo Simulation 81
6.1 Preliminary Settings . . . 82
6.2 Simulation Results . . . 83

7 Conclusions 87

A Toolflow setup guide for Smart Industry use case 89
A.1 Main Steps . . . 89
A.1.1 Docker_arch . . . 89
A.1.2 Orchestrator . . . 90
A.1.3 Webserver (orchestrator frontend) installation . . . 90
A.1.4 Run the webserver . . . 91

B Kinematics 92
B.1 DH convention . . . 92
B.2 Inverse kinematics computation . . . 96

B.3 ***ZYZ*** *Euler Angles* 98
B.4 Anthropomorphic arm configurations 100
B.5 Detailed steps for e.DO connection 101

C Perception Pipeline 102
C.1 Code Snippets 102
C.1.1 Code snippet of the addCylinder() function 102
C.1.2 Code snippet of the Update Planning Scene Control Sequence 106
C.2 Quaternions and Pose Estimation 107
C.2.1 Axis Angle representation 107
C.2.2 Quaternion 108
C.2.3 Pose and Orientation for the Collision Object 109

Bibliography 111

List of Tables

1.1 Use cases reference baseline toolflow projects 4
1.2 Words and frequencies in speech commands datset V2 6
1.3 CNN architecture for cnn-trad-fpool3 ([8]) 10
1.4 CNN architecture for cnn-one-stride4 ([8]) 11
1.5 Performance of CNN variants on the Raspberry Pi ([11]) 11
1.6 Scenario's commands and their relative actions 14

2.1 Cobot specifications 20
2.2 DH kinematics parameters 22
2.3 e.DO links in mm 22

List of Figures

1.1 SensorTile functional block diagram 8
1.2 Diagram of MFCC Derivation Process ([7]) 9
1.3 Structure of the KWS-NET architecture ([7]) 10
1.4 Depthwise separable CNN architecture ([10]) 12
1.5 e.DO demo set-up 14
1.6 SensorTile configuration with Expansion cradle and Nucleo Board . 15
1.7 SensorTile output (PuTTy) 16

2.1 Available configurations for e.DO Cobot 18
2.2 e.DO geometry and workspace 19
2.3 Description of the position and orientation of the end-effector frame 19
2.4 Coordinate transformations in an open kinematic chain 21
2.5 DH convention link frames 21
2.6 ROS File System Level 26

3.1 Internal structure of the Intel RealSense D435i camera 31
3.2 Camera reference triad chosen for the software development (right triad) 32
3.3 Point Cloud before (left) and after (right) filtering 34
3.4 Point Cloud before (left) and after (right) Statistical Outlier Removal 34
3.5 Point Cloud before (left) and after (right) planar surface removal . . 35
3.6 Cloud clusters before (left) and after (right) polynomial reconstruction 37
3.7 Normals computation with small scale (left) and large scale (right) . 38
3.8 Point Cloud models 41
3.9 Graphical representation of the center point and height estimation . 43
3.10 HSV cone and parameters definition 44
3.11 Simulation configuration before (left) and after processing(right) . . 48
3.12 3D Object Recognition wrt Model1 for cluster 1 49
3.13 3D Object Recognition wrt Model2 for cluster 1 49
3.14 3D Object Recognition wrt Model3 for cluster 1 49
3.15 3D Object Recognition wrt Model4 for cluster 1 50

3.16 3D Object Recognition wrt Model5 for cluster 1 50
3.17 3D Object Recognition wrt Model6 for cluster 1 51
3.18 3D Object Recognition wrt Model7 for cluster 1 51
3.19 3D Object Recognition wrt Model1 for cluster 2 51
3.20 3D Object Recognition wrt Model1 for cluster 3 52

4.1 Pick Place planning scene configuration 55
4.2 Grasp 56
4.3 Gripper pose at rest 57
4.4 Gripper pose at grasp 57
4.5 Pick Place planning scene with e.DO at rest 58
4.6 Pre-Grasp approach 59
4.7 Post-Grasp retreat 60
4.8 Pre-place approach 61
4.9 Place 61
4.10 Post-place retreat 62
4.11 Results for grasps generation and filtering 63
4.12 Pre-grasp approach 64
4.13 Grasp 64
4.14 Post-grasp lift 65
4.15 Post-grasp retreat 65
4.16 Conversion from linear to angular finger opening 67

5.1 Demo planning scene setup 71
5.2 Demo planning scene setup with random generator 72
5.3 Demo planning scene setup with perception generator 74
5.4 Faulty Placing 78
5.5 Correct Placing 79

6.1 Start simulation: cylinders have been sensed and added to the planning scene. 83
6.2 First object Pick. 83
6.3 First object Place. 84
6.4 Second object Pick. 84
6.5 Second object Place. 85
6.6 Third object Pick. 85
6.7 Third object Place. 86
6.8 Simulation end: The robot moves back to its rest position. New objects are sensed and added to the planning scene, waiting for a new computation. 86

B.1 Denavit-Hartemberg kinematic parameters 92

B.2 Representation of ZYZ Euler Angles 98
B.3 The four configurations of an anthropomorphic arm compatible with a given wrist position 100

C.1 Rotation of an angle about an axis 108
C.2 RPY cam configuration wrt scene 110

Acronyms

Rviz
: 3D visualization tool for ROS

ROS
: Robot Operating System

DH
: Denavit-Hartemberg

KWS
: KeyWord Spotting

CNN
: Convolutional Neural Network

RNN
: Recurrent Neural Network

DL
: Deep Learning

CV
: Computer Vision

IoT
: Internet of Things

MFCC
: Mel-Frequency Cepstral Coefficients

PLC

Programmable Logic Controller

SoC

System on Chip

FPS

Frames Per Second

PCL

Point Cloud Library

ROI

Region Of Interest

RANSAC

RANdom SAmple Consensus

VCS

Version Control System

svn

Subversion

hg

Mercurial

GUI

Graphic User Interface

API

Application Programming Interface

LRF

Local Reference Frame

MLS

Moving Least Square

ICP

Iterative Closest Point

SHOT

Signature of Histograms of OrienTations

OMP

OpenMP standard

FLANN

Fast Library for Approximate Nearest Neighbor

BOARDLRF

BOrder Aware Repeatable Directions algorithm for Local Reference Frame estimation

ALOHA

Software framework for runtime-Adaptive and secure deep Learning On Heterogeneous Architectures

Chapter 1

Deep Learning at the Edge using ALOHA Toolflow

Deep Learning (DL) algorithms are an extremely promising instrument in artificial intelligence, achieving very high performance in numerous recognition, identification, and classification tasks. To foster their pervasive adoption in a vast scope of new applications and markets, a step forward is needed towards the implementation of the on-line classification task (called inference) on low-power embedded systems, enabling a shift to the edge computing paradigm. Nevertheless, when DL is moved at the edge, severe performance requirements must coexist with tight constraints in terms of power/ energy consumption, posing the need for parallel and energy-efficient heterogeneous computing platforms. Unfortunately, programming for this kind of architectures requires advanced skills and significant effort, also considering that DL algorithms are designed to improve precision, without considering the limitations of the device that will execute the inference. Thus, the deployment of DL algorithms on heterogeneous architectures is often unaffordable for SMEs and midcaps without adequate support from software development tools.

The main goal of ALOHA is to facilitate implementation of DL on heterogeneous low-energy computing platforms. To this aim, the project will develop a software development tool flow, automating:

- algorithm design and analysis;
- porting of the inference tasks to heterogeneous embedded architectures, with optimized mapping and scheduling;
- implementation of middleware and primitives controlling the target platform, to optimize power and energy savings.

During the development of the ALOHA tool flow, several main features will be addressed, such as architecture-awareness (the features of the embedded architecture will be considered starting from the algorithm design), adaptivity, security, productivity, and extensibility. ALOHA will be assessed over three different use-cases, involving Surveillance of Critical Infrastructures, Smart Industry automation, and Medical application domains[1].

1.1 Toolflow Description and Application

As the ALOHA project evolves, the main goal was to define, for each use case, the first complete set of inputs needed by the Toolflow. Here, the interest does not just fall on describing the task that the Toolflow should perform, but also on providing all the necessary data for the task to be completed.

The main idea here is that the use case providers can test the capability of the Toolflow to provide design points that are coherent with the defined constraints. Another goal is to provide the use case partners with the possibility to integrate the Toolflow in their workflows. Later on, the use case providers will define other projects based on the different aspects of each use case.

Table 1.1 outlines the baseline projects for each use case. Following elements are defined:

- *Dataset available*: determines which dataset is provided by the use case. The labels of the dataset define the task expected to be accomplished for the optimized DNN algorithm;
- *Initial reference algorithm*: defines which Deep Learning reference network will be used as starting point by the Toolflow;
- *Reference implementation available*: defines if there is already an implementation of the initial algorithm that could be used as reference for analyzing the Toolflow results;
- *Prospective target platform*: the embedded hardware platform being targeted by the baseline project. This will define which hardware description will be used;
- *Constraints*: define the main constraints of the project regarding performance, accuracy and security level. Following constraints are defined:
  - *Performance*: Defines the inference time for each sample. A sample is one single data element, as for instance an image. The performance constraint is defined in milliseconds per sample (ms/sample);

  - *Security*: The security level expected. Three Values are available, with the following meaning: HIGH means that accuracy does not drop more than 33%, MED: accuracy drops more than 33% but no more than 66%, LOW: accuracy drops more than 66%;
  - *Accuracy*: The accuracy expected for the data set in percent. Note that the actual measurement method depends on the task and depends on the implementation of the training engine. Each use case provider will explain the metric being used in their corresponding section;
  - *Power consumption*: The energy needed to calculate the inference for each sample in millijoule per sample (mJ/sample). For each constraint, a priority level from 1 to 4, where 1 is the highest priority, will be selected. This will be used by the Design Space Exploration engine for the algorithm optimization

- *Mandatory non-DL tasks*: here the use case determines which algorithm must be performed so that the results can be generated by the Toolflow and used by the use case application. Examples are mandatory pre/post processing algorithms.

ALOHA Use Cases	Surveillance	Medical	Audio
Dataset available	Yes (based on Cityscape and COCO)	Yes	Yes (Google Speech Command Dataset v2)
Initial reference algorithm	Tiny YOLO	U-NET	KWS-NET (cnn-tred-fpool3)
Reference implementation available	Yes (on NCS)	Yes (on PC)	Yes (on SensorTile)
Perspective target platform	NEURAghe/Orlando	NEURAghe (ZC706)	Sensortile/Orlando
Constraints - performance	125 ms/sample	Under 3 minutes on NEURAghe. Under 1 minute on the defined high end PC	250 ms/sample
Constraints - security and threat model	HIGH	LOW	HIGH
Constraints - accuracy	> 60% AP	Sensitivity>90% Specificity>85%	>90%
Constraints - power consumption	1250 mJ/sample (considering 10W, 8FPS)	None	450 mJ/sample
Mandatory non-DL task	NMS for YOLO	MaxQ.AI C++ Dlls	MFCC

Table 1.1: Use cases reference baseline toolflow projects

1.2 Command Recognition in Smart Industry Applications

In the smart industry use case, that is the starting point of this thesis, the aim is to develop an embedded keyword spotting system that would activate/deactivate a PLC-controlled tooling machinery or collaborative robot in an industrial environment, without relying on a cloud backend.

1.2.1 Baseline Dataset Preparation

Google Spech Commands Datset Description and Organization

Considering the focus on keyword spotting and the license type, the Google Speech Command Dataset has been chosen for model training and comparative test of different models in ALOHA.

The Google Speech Command dataset [2] is a set of one-second .wav audio files, each containing a single spoken English word. This data set is designed to help train simple machine learning models and it was released under the Creative Commons BY 4.0 license. Version 1 of the data set was released on August 3rd, 2017 and contained 64,727 audio files. Version 2 includes 105,829 audio files, released on April 11th, 2018.

The words included in the dataset are from a small set of commands and are spoken by a variety of different speakers using crowdsourcing. The goal was to gather examples of people speaking single-word commands, rather than conversational sentences, so they were prompted for individual words over the course of a five-minute session.

The original audio files were collected in uncontrolled locations by people around the world without any quality requirements or control over the recording equipment or environment. An open-source web-based application that records utterances is available [3]. The data was captured in a variety of formats, and then converted to a 16-bit little-endian PCM-encoded WAVE file at 16 kHz rate. The audio was then trimmed to a one second length to align most utterances, screened for silence or incorrect words, and arranged into folders by label.

In version 1 of the dataset twenty core command words were recorded, with most speakers saying each of them five times. They include the digits zero to nine and the words "Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", and "Go", useful as commands in IoT or robotics applications. In version 2 of the dataset, four more command words were added; “Backward”, “Forward”, “Follow”, and “Learn”. To help distinguish unrecognized words, and ignoring speech that doesn’t contain triggers, also the following ten auxiliary words were included: "Bed", "Bird", "Cat", "Dog", "Happy", "House", "Marvin", "Sheila", "Tree", and "Wow". Some of these,

such as “Tree”, were picked because they sound similar to target words and would be good tests of a model’s discernment. The complete list of keywords, broken into the categories and frequencies, is shown in Table 1.2. In the project version 2 of the dataset is used.

World	Number of Utterances
Backward	1.664
Bed	2.014
Bird	2.064
Cat	2.031
Dog	2.128
Down	3.917
Eight	3.787
Five	4.052
Follow	1.579
Forward	1.557
Four	3.728
Go	3.880
Happy	2.054
House	2.113
Learn	1.575
Left	3.801
Marvin	2.100
Nine	3.934
No	3.941
Off	3.745
On	3.845
One	3.890
Right	3.778
Seven	3.998
Sheila	2.022
Six	3.860
Stop	3.872
Three	3.727
Tree	1.759
Two	3.880
Up	3.723
Visual	1.592
Wow	2.123
Yes	4.044
Zero	4.052

Table 1.2: Words and frequencies in speech commands datset V2

Files are organized into folders, with each directory name labelling the word that is spoken in all the contained audio files. No details were kept of any of

the participants' age, gender, or location, and random IDs were assigned to each individual. These IDs are stable though, and encoded in each file name as the first part before the underscore. If a participant contributed multiple utterances of the same word, these are distinguished by the number at the end of the file name. For example, the file path 'happy/3cfc6b3a_nohash_2.wav' indicates that the word spoken was "happy", the speaker's ID was "3cfc6b3a", and this is the third utterance of that word by this speaker in the data set.

A key requirement for keyword spotting in real products is to cope with noisy environments and distinguishing between audio that contains speech, and clips that contain non-speech sounds. To help train and test this capability, the "_background_noise_" folder, included in the target dataset, contains a set of longer audio clips that are either recordings or mathematical simulations of noise.

Considerations

The neural network models are trained to classify the incoming audio into one of the 10 keywords - "Yes", "No", "Up", "Down", "Left", "Right", "On", "Off", "Stop", "Go", along with "silence"(i.e., nowordspoken) and "unknown" words, which is the remaining 25 keywords from the dataset. The dataset is split into training, validation and test sets in the ratio of 80:10:10 while making sure that the audio clips from the same person stay in the same set.

The audio clips haven't been separated into training, test, and validation sets explicitly, but by convention a hashing function is used to stably assign each file to a set. The results of running this over the current set are included in the dataset archive as *validation_list.txt* and *testing_list.txt*. These text files contain the paths to all the files in each set, with each path on a new line. Any files that aren't in either of these lists can be considered to be part of the training set.

This format is compliant with the .txt encoding format, as described in the dataset standardization representation [4], and in the next period it has been considered to extend the dataset by recording background noise specific for the target deployment environment and/or custom commands.

1.2.2 Target Embedded Hardware

In order to realize an initial reference implementation, the SensorTile has been selected, a tiny IoT module based on STM32L4 ultra low-power microcontroller with Arm-Cortex M4 core and featuring a development kit that simplifies prototyping, evaluation and development of innovative solutions.

This allows to realize an early demonstrator with a simpler and commercial-grade environment, and to reuse the results in the final demonstrator with the target environment.

In the following, the characteristics of the platform [5] will be outlined:

SensorTile Key Features

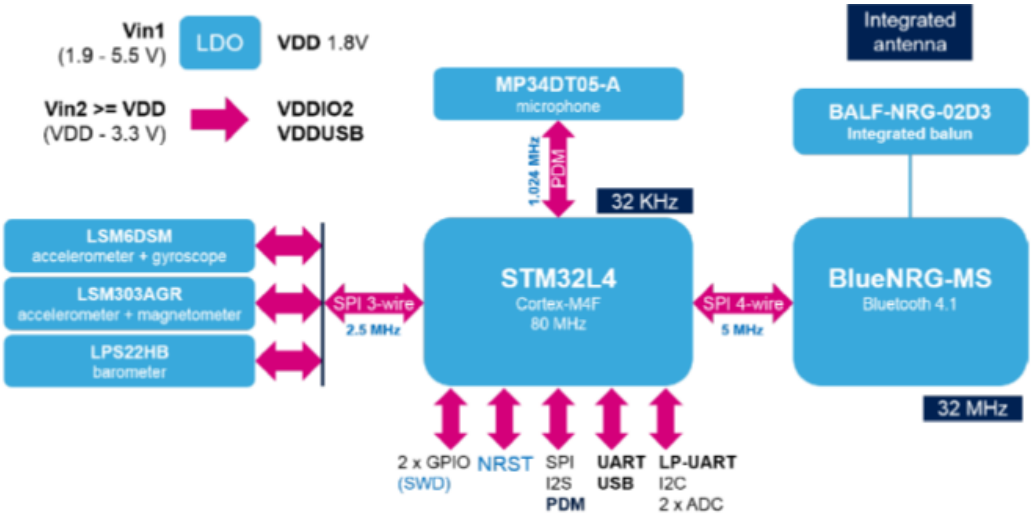

Figure 1.1: SensorTile functional block diagram

Highly integrated and very compact module that can be plugged into form factor prototypes to add motion, audio, environmental sensing and Bluetooth low energy connectivity.

- Host: Arm Cortex-M4 32-bit;
- Frequency: 80 MHz;
- On chip RAM: 128 KB, Flash: 1 MB;
- MEMS audio sensor omnidirectional digital microphone.

1.2.3 Baseline CNN Algorithm

Input Data Preparation

To be suitable for CNN processing, speech signals need to be organized as a number of feature maps. This is a term borrowed from image-processing applications, in which it is intuitive to organize the input as a two-dimensional (2-D) array. For speech-processing applications we first need to calculate Mel-Frequency Cepstral Coefficients (MFCC) to extract spectral features [6].

Figure 1.2: Diagram of MFCC Derivation Process ([7])

As shown in 1.2, the first step is to apply a pre-emphasis filter on the signal to amplify the high frequencies and then to split the signal into short-time frames. The rationale behind this step is that frequencies in a signal change over time, so in most cases it doesn't make sense to do the Fourier transform across the entire signal since it can cause a lose of the frequency contours of the signal over time. The input speech signal of length L is framed into overlapping frames of length l with a stride s, giving a total of $T = \frac{L-l}{s} + 1$ frames. Typical frame sizes in speech processing range from 20 ms to 40 ms with 50% (+/-10%) overlap between consecutive frames. Assuming 30 ms analysis window, and shifting stride = 10 ms, since audio signal sample is 1 s each, it would results in $\frac{1000-30}{10} + 1 = 98$ time frames.

After windowing, Fast Fourier Transformation (FFT) is calculated for each frame to obtain the frequency features, and the logarithmic Mel-Scaled filter bank (typically 40 filters) is applied to the Fourier transformed frames. The Mel-scale aims to mimic the non-linear human ear perception of sound, by being more discriminative at lower frequencies and less discriminative at higher frequencies. The last step is to calculate Discrete Cosine Transformation (DCT) to decorrelate the filter bank coefficients and yield a compressed representation as Mel-Frequency Cepstrum Coefficients (MFCCs). From each frame, F speech features are extracted, generating a total of TxF features for the entire input speech signal of length L.

KWS-NET Architecture

The initial CNN model selected for the smart industry use-case is referred in this document as KWS-NET and it corresponds to the 'cnn-trad-fpool3' model presented in [8] and illustrated in Figure 1.3.

The key layers are: Convolutional (Conv) layer (multiple convolution filters to obtain different features), Pooling layer (down-sampling by taking max operation to reduce the amount of parameters and computation in the network, and hence control overfitting), Dropout layer (only keep a neuron active with some probability p, or set it to zero otherwise to control overfitting), Linear low-rank (Lin) layer (perform linear multiplication and addition to transfer the output of Conv layer to discrete nodes, reduce parameters and computation, control overfitting), and Fully-connected (FC) layer (preserve full information, or make the final softmax

prediction).

This architecture allows to keep the total number of parameters below 250k, however a main issue is the huge number of multiplies in the convolutional layers, which get exacerbated in the second layer because of the 3-dimensional input, spanning across time, frequency and feature maps, as shown in Table 1.3.

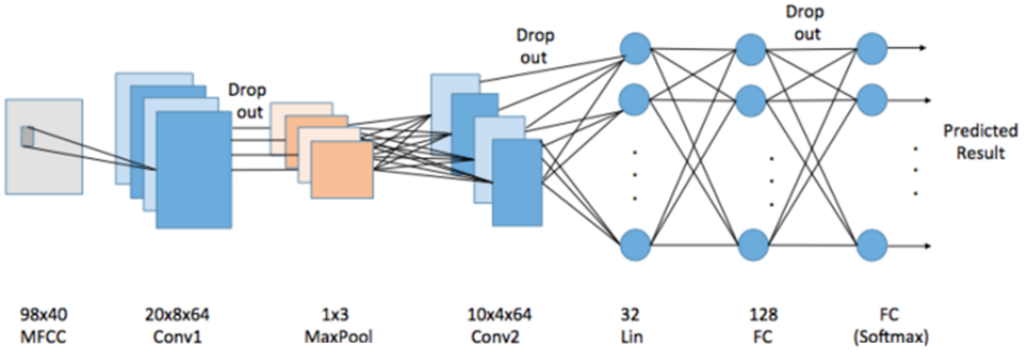

Figure 1.3: Structure of the KWS-NET architecture ([7])

Type	m	r	n	p	q	Par.	Mul
Conv	20	8	64	1	3	10.2K	4.4M
Conv	10	4	64	1	1	164.8K	5.2M
Lin	-	-	32	-	-	65.5K	65.5K
FC	-	-	128	-	-	4.1K	4.1K
SoftMax	-	-	4	-	-	0.5K	0.5K
Tot	-	-	-	-	-	**244.2K**	**9.7M**

Table 1.3: CNN architecture for cnn-trad-fpool3 ([8])

Other CNN Architecture Considered

In order to limit the number of multiplies, [8] proposed an alternative architecture with one convolutional layer rather than two, and the time filter span all of time. The output of this convolutional layer is passed to a linear low-rank layer, and then two FC layers. Moreover instead then pool in frequency stride the filter in frequency by four. (For a stride of one it moves across and down a single neuron. With higher stride values, it moves large number of neuron at a time and hence produce smaller output volumes). As shown in Table 1.4 this architecture, called 'cnn-one-stride4', cut the multipliers number by a factor of ten, compared to 'cnn-trad-fpool3'.

Type	m	r	n	p	q	Par.	Mul
Conv	32	8	186	1	4	47.6K	428.5K
Conv	-	-	32	-	-	19.8K	19.8K
Lin	-	-	128	-	-	4.1K	4.1K
FC	-	-	128	-	-	16.4K	16.4K
SoftMax	-	-	4	-	-	0.5K	0.5K
Tot	-	-	-	-	-	**88.4K**	**469.3K**

Table 1.4: CNN architecture for cnn-one-stride4 ([8])

Another promising architecture considered for the implementation of the smart industry use-case is the DS-CNN (depthwise separable convolution) model, which has been recently proposed as an efficient alternative to the standard 3-D convolution operation in the area of computer vision [9]. By decomposing the standard 3-D convolutions into 2-D convolutions followed by 1-D convolutions (see Figure 1.4), DS-CNNs result more efficient in terms of computational requirements (i.e. number of parameters, operations), which makes them suitable for deployment in resource-constrained devices [10].

1.2.4 Baseline Constraints

As shown in Table 1.5, cnn-one-stride4 presents a Latency/sample and Energy/sample sensibly lower than the cnn-trad-fpool3 but also an Accuracy performance much worse, therefore it will be considered only in case the constraints cannot be met with the cnn-trad-fpool3 and a more compact model is required.

Model	Test Accuracy	Latency/q (ms)	Energy/q mJ	Peak Power (W)
cnn-trad-fpool3	89.43%	227	431	2.20
cnn-one-stride4	70.28%	40	28	0.99
Feature extraction only	-	31	19	0.80

Table 1.5: Performance of CNN variants on the Raspberry Pi ([11])

Power Comsumption

Due to always-on nature, a KWS application presents highly constrained power budget if batteries are used. However, in industrial settings, it is often possible to plug the system on a charger. Therefore, for this parameter, a value of 450 mJ/sample and priority 3 have been set.

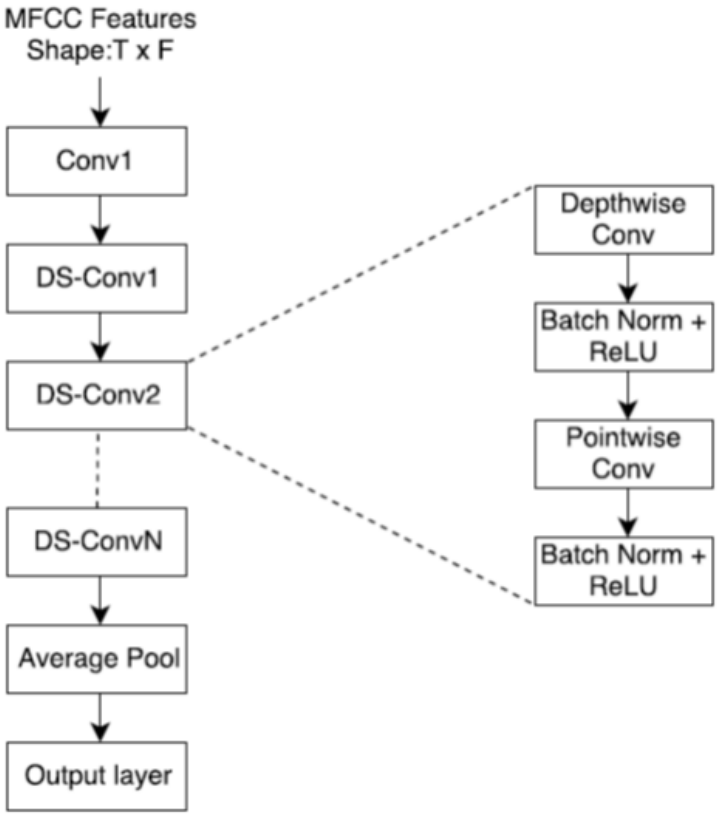

Figure 1.4: Depthwise separable CNN architecture ([10])

Security

In a smart industry use-case, devices must work in rugged, often noisy environments. Most of the input will be silence or background noise, not speech, so false positives on those must be minimized. Moreover, most of the input that is speech will be unrelated to the voice interface, so the model should be unlikely to trigger on arbitrary speech. Since some commands are phonetically similar, it is also important to avoid misunderstanding. For these reasons, this constraint value has been set as HIGH and with priority level equal to 2.

Performance

The voice control in an industrial environment task requires real time response. In an embedded implementation there are no network delays due to cloud back-end, but limited memory footprint and compute resources. However the priority

could be relaxed as the task requires commands recognition and not continuous speech.Therefore, for this parameter a value of 250 ms/q and priority 4 have been set.

Accuracy

High accuracy is required to avoid operator frustration. The metric that was considered is the Top-One, as described in []. The rates reported in literature for this task are around 85-95%. As this parameter is of paramount importance for the system acceptance it has been set as >90% and the priority as 1.

1.3 System Design and Implementation

The goal of Smart Industry use case would be the ability to operate machinery through speech recognition. **REPLY** aims at developing an embedded keyword spotting system that would activate/deactivate a collaborative robot in an industrial environment, without relying on a cloud backend. In order to make the keyword spotting system adaptable to different industrial scenarios, a very popular type of interface has been chosen to dialogue between the recognition system and the system to be controlled: a serial standard interface on which the recognition system sends commands to the industrial machinery.

Practically the cobot can be controlled in two different ways and it is possible to switch among them through the KWS command ***yes***. Below the differences are highlighted:

- ***joint*** mode. The cobot is controlled through MoveIt sending directly a target angle to joints. Six different commands are available in this configuration:
  - ***up***, ***down***, ***left*** and ***right***, that correspond to the four poses that can be seen in Figure 1.5;
  - ***off*** that corresponds to the cobot idle position (candle shape);
  - ***on*** that shakes the robot forward and backward;
- ***spatial*** mode. The cobot is controlled in terms of cartesian coordinates x, y and z through the MoveIt inverse kinematics. Also in this case six different commands are available:
  - ***up*** and ***down*** to increase and decrease respectively, the z coordinate;
  - ***left*** and ***right*** to increase and decrease respectively, the x coordinate;
  - ***on*** and ***off*** to increase and decrease respectively, the y coordinate;

In Figure 1.5 and Table 1.6, a simple representation of the scenario to develop and its commands and relative actions are shown respectively:

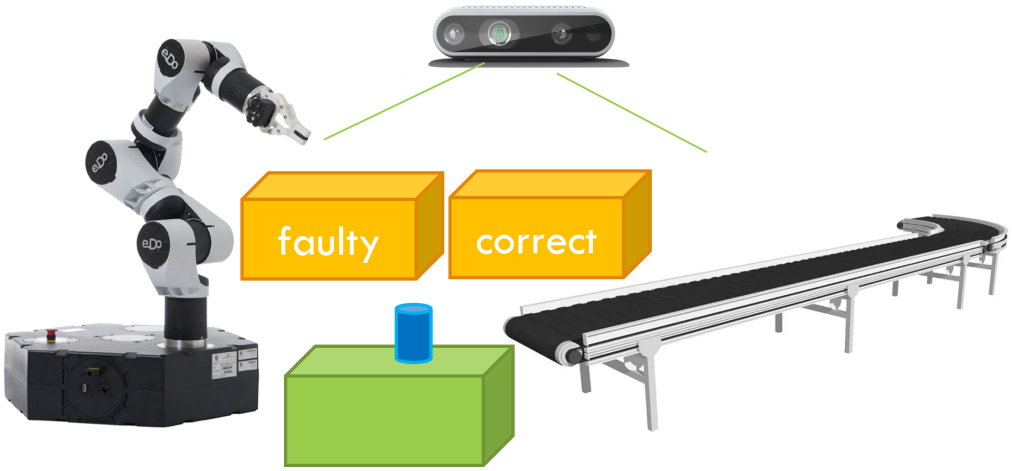

Figure 1.5: e.DO demo set-up

Command	Action
Yes	Move object to *correct* area
No	Move object to *faulty* area

Table 1.6: Scenario's commands and their relative actions

It is important to notice that:

- The objects of interest are assumed to be cylinders of known dimensions but, with some approximation, can be generalized to other shapes;
- The relative poses of cobot, camera, objects initial area, *correct* and *faulty* boxes must be identified and fixed in space;
- The management of the object orientation must be verified.

Set-up KWS on SensorTile

To set-up the SensorTile board, the instructions reported in the "STEVAL-STLKT01V1 Quick start guide" available on STMicroelectronics web site [12] has been followed.

The SensorTile has been connected to the Expansion Cradle and Nucleo board (see Figure 1.6) and both boards to the PC through two micro USB cables. To build and deploy the embedded software, we used the System WorkBench Integrated Development Environment for STM32 [13].

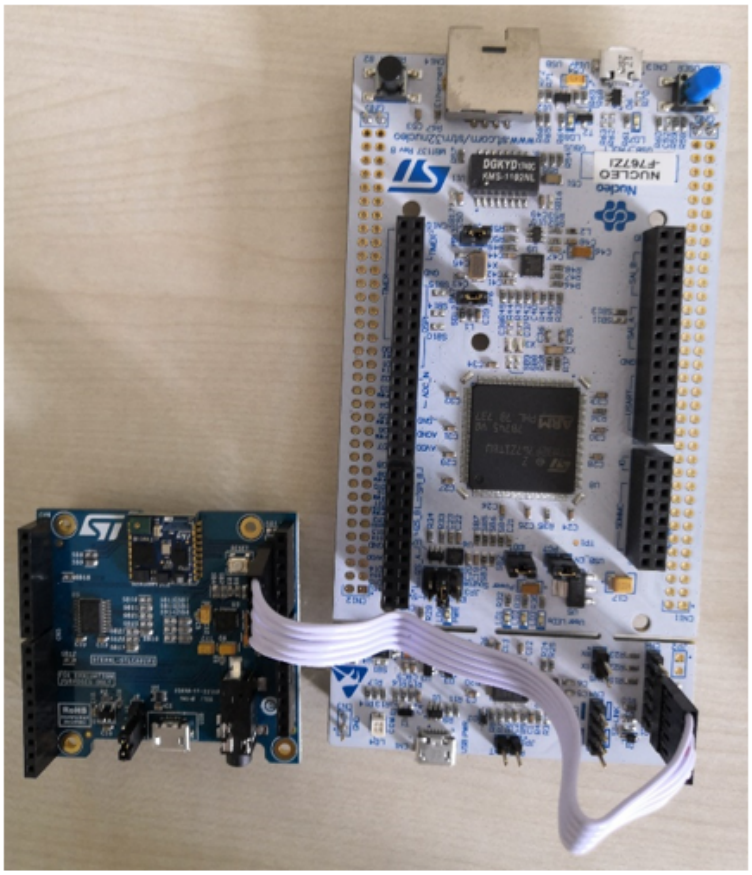

Figure 1.6: SensorTile configuration with Expansion cradle and Nucleo Board

The data transmitted by the SensorTile board over serial USB connection to the personal computer, are available through PuTTy[14] SSH and telnet client, as shown in Figure 1.7.

As can be seen in Figure 1.7, the application captures audio signals thorough the SenorTile on-board microphone, processes them and returns the detected keyword

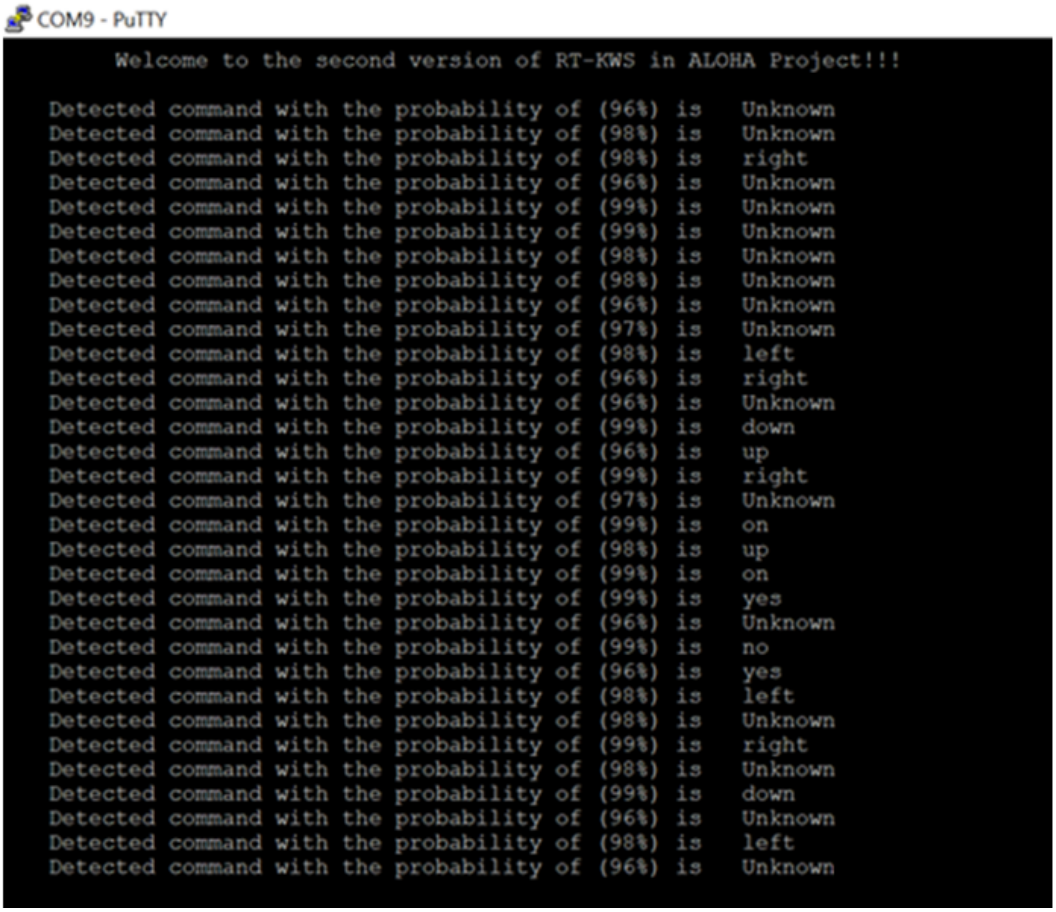

Figure 1.7: SensorTile output (PuTTy)

and its associated probability (the probability threshold has been set to 95%).

1.4 References

For any reference to experienced models, power consumption, KWS on μC, CNNs for small footprint and speech commands, please take a look respectively to [11], [10], [15], [8], [7].

Chapter 2

Robotic Arm Control

2.1 e.Do Cobot

e.DO Cobot [16] is a modular, multi-axis articulated (anthropomorphic with 6 DOFs) Educational Robot, with an integrated open-source intelligence, produced by COMAU. It is available in different versions and configurations:

1. e.DO with 6 central axis;
2. e.DO with 6 side axis;
3. e.DO with 6 central axis, with gripper;
4. e.DO with 6 side axis, with gripper.

2.2 Mechanical Characteristics and Kinematics

The anthropomorphic manipulator e.DO has 6 degrees of freedom, due to the six revolute joints, to which it is possible to add another DOF thanks to the gripper. The geometry of the mechanical system and its workspace are shown in Figure 2.2, meanwhile in table 2.1 are listed the main dimensional and working characteristics.

2.2.1 Direct Kinematics

The direct kinematic problem consists in the identification of the end-effector pose and orientation as a function of the joint variables. Considering that the pose of a body in space is described by the position vector of the origin and the unit

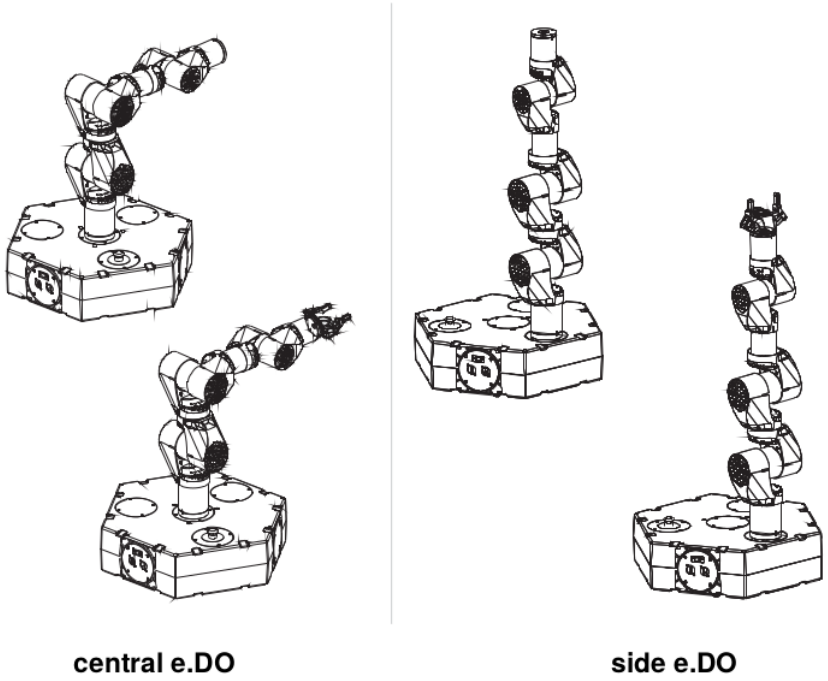

Figure 2.1: Available configurations for e.DO Cobot

vectors of a frame attached to the body, considering the frame O_b-$x_b y_b z_b$, the direct kinematic functions can be expressed by the following homogeneous matrix

$$\boldsymbol{T}_e^b(\boldsymbol{q}) = \begin{bmatrix} \boldsymbol{n}_e^b(\boldsymbol{q}) & \boldsymbol{s}_e^b(\boldsymbol{q}) & \boldsymbol{a}_e^b(\boldsymbol{q}) & \boldsymbol{p}_e^b(\boldsymbol{q}) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

where q is the *(n x 1)* vector of joint variables, $\boldsymbol{n_e}$, $\boldsymbol{s_e}$, $\boldsymbol{a_e}$ are the unit vectors of a frame attached to the end-effector and $\boldsymbol{p_e}$ is the position vector of the origin of such a frame with respect to the origin of the base frame O_b-$x_b y_b z_b$, as shown in Figure 2.3. It is important to notice that $\boldsymbol{n_e}$, $\boldsymbol{s_e}$, $\boldsymbol{a_e}$ and $\boldsymbol{p_e}$ are function of $\boldsymbol{q}$.

Considering the open-chain structure of the e.DO cobot, the computation of direct kinematics is derived by the description of kinematic relationship between consecutive links, from which it is possible to obtain the overall description in a recursive fashion. To this purpose it is worth defining a coordinate frame for each link, considering that the coordinate transformation describing pose and orientation of the end-effector with respect to the first frame is given by

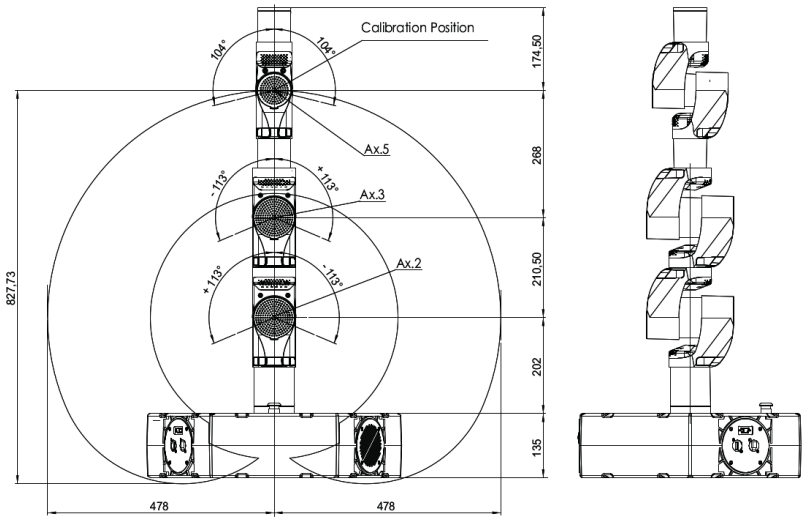

Figure 2.2: e.DO geometry and workspace

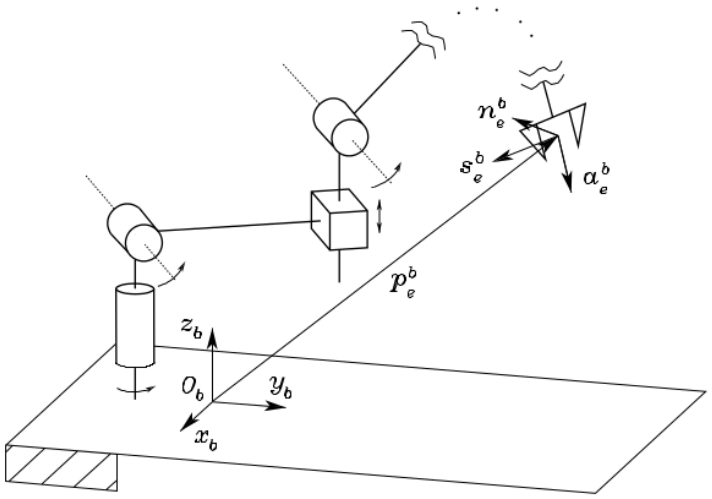

Figure 2.3: Description of the position and orientation of the end-effector frame

$$\boldsymbol{T}_n^0(\boldsymbol{q}) = \boldsymbol{A}_1^0(\boldsymbol{q_1})\boldsymbol{A}_2^1(\boldsymbol{q_2})\dots\boldsymbol{A}_n^{n-1}(\boldsymbol{q_n})$$

where n is equal to 6. In this way the computation of direct kinematics function is recursive and can be obtained in a systematic way, by simple products of homogeneous transformation matrices $\boldsymbol{A_i}^{\boldsymbol{i\text{-}1}}(q_i)$ (for i=1, . . . , n), each of which is a

Specifications	Value
Number of axis	6
Max payload	1 Kg
Max reach	478 mm
Stroke (Speed) Axis1 Axis2 Axis3 Axis4 Axis5 Axis6	 +/-180° (38°/sec) +/-113° (38°/sec) +/-113° (38°/sec) +/-180° (56°/sec) +/-104° (56°/sec) +/-2700° (56°/sec)
Total weight	11.1 kg
Robot arm weight	5.4 kg
Structure material	Ixef 1022
Power source	Universal external power source with 12 V power adapter
Connectivity	1 external USB port 1 RJ45 Ethernet 1 DSub-9 Serial Port
Motherboard	Raspberry Pi running Raspbian Jessie
ROS	Kinematic Kame
Control logic	Proprietary open-source **e.DO**
Additional Features	External emergency push button

Table 2.1: Cobot specifications

function of a single joint variable.
So the actual coordinate transformation describing the position and orientation of the end-effector frame with respect to the base frame can be computed as

$$\boldsymbol{T}_b^e(\boldsymbol{q}) = \boldsymbol{T}_0^b \boldsymbol{T}_n^0(\boldsymbol{q}) \boldsymbol{T}_e^n$$

where $\boldsymbol{T_0}^{\boldsymbol{b}}$ and $\boldsymbol{T_e}^{\boldsymbol{n}}$ typically are two constant homogeneous transformations describing the position and orientation of Frame 0 with respect to the base frame, and of the end-effector frame with respect to Frame n, respectively. A general case of the direct kinematic computation can be seen in Figure 2.4

e.DO is characterized by 6 degrees of freedom, thanks to the presence of the six independent revolute joints. Applying the convention described in Appendix B.1, it is possible to describe the geometric transformation from triad i to triad $i+1$ only by the four DH parameters:

1. $a_i \rightarrow$ Distance between $\boldsymbol{O_i}$ and $\boldsymbol{O_i}$';
2. $d_i \rightarrow$ Coordinate of $\boldsymbol{O_i}$' along $\boldsymbol{z_{i-1}}$;

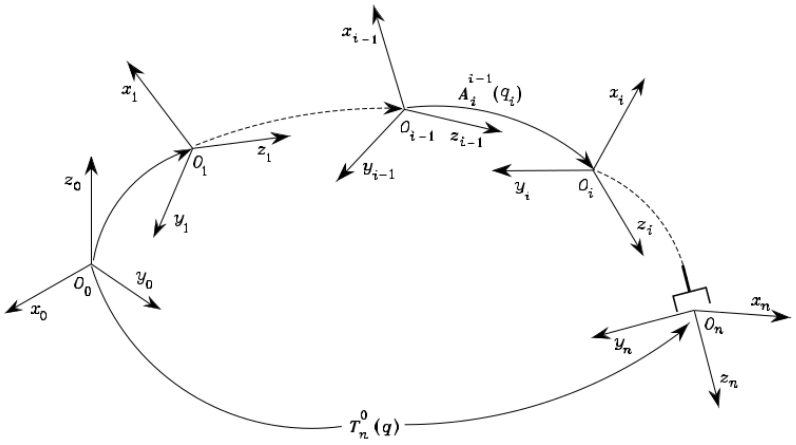

Figure 2.4: Coordinate transformations in an open kinematic chain

3. $\alpha_i \rightarrow$ Angle between axis $\boldsymbol{z_{i-1}}$ and $\boldsymbol{z_i}$ about axis $\boldsymbol{x_i}$ (positive for counter-clockwise rotations);

4. $\theta_i \rightarrow$ Angle between axis $\boldsymbol{x_{i-1}}$ and $\boldsymbol{x_i}$ about axis $\boldsymbol{z_{i-1}}$ (positive for counter-clockwise rotations).

Once all the link frames have been fixed as in figure 2.5

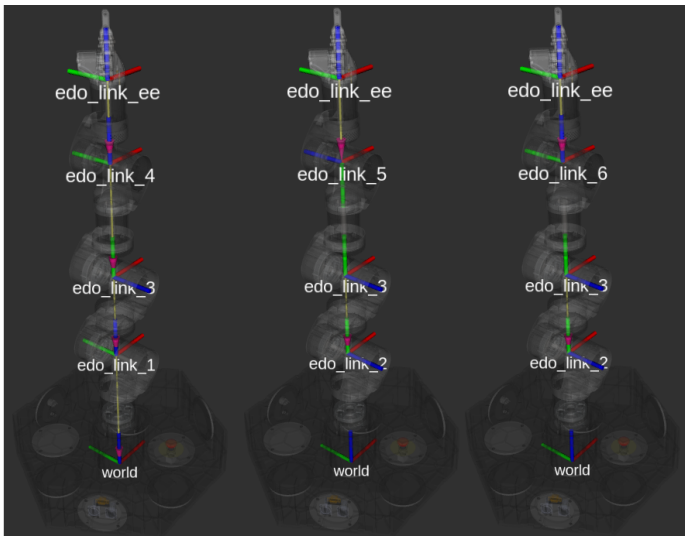

Figure 2.5: DH convention link frames

we obtain DH parameters reported in table 2.2

i	a_i	α_i	d_i	θ_i
1	0	$\pi/2$	0	θ_1
2	L_1	0	0	θ_2
3	L_2	$-\pi/2$	0	θ_3
4	0	$-\pi/2$	0	θ_4
5	0	$\pi/2$	0	θ_5
6	0	0	L_3	θ_6

Table 2.2: DH kinematics parameters

i	L_i
1	210.5
2	268
3	174.5

Table 2.3: e.DO links in mm

Once the DH parameters are available, each couple link-joint can be described as a coordinate transformation between two reference frames associated to the joints. Since the $\boldsymbol{X_n}$ axis is oriented along the common normal between axis $\boldsymbol{Z_{n-1}}$ and $\boldsymbol{Z_n}$, the homogeneous transformation matrix is defined as a series of two consecutive roto-translations

$$\boldsymbol{T}_n^{n-1} = \boldsymbol{Trasl}_{z_{n-1}}(d_n)\boldsymbol{Rot}_{z_{n-1}}(\theta_n)\boldsymbol{Trasl}_{x_n}(a_n)\boldsymbol{Rot}_{x_n}(\alpha_n)$$

where:

$$\boldsymbol{Trasl}_{z_{n-1}}(d_n) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{Rot}_{z_{n-1}}(\theta_n) = \begin{pmatrix} \cos\theta_n & -\sin\theta_n & 0 & 0 \\ \sin\theta_n & \cos\theta_n & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{Trasl}_{x_n}(a_n) = \begin{pmatrix} 1 & 0 & 0 & a_n \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{Rot}_{x_n}(\alpha_n) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_n & -\sin\alpha_n & 0 \\ 0 & \sin\alpha_n & \cos\alpha_n & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

that results in the complete homogeneous transformation matrix

$$\boldsymbol{T}_n^{n-1} = \begin{pmatrix} \cos\theta_n & -\sin\theta_n\cos\alpha_n & \sin\theta_n\sin\alpha_n & a_n\cos\theta_n \\ \sin\theta_n & \cos\theta_n\cos\alpha_n & -\cos\theta_n\sin\alpha_n & a_n\sin\theta_n \\ 0 & \sin\alpha_n & \cos\alpha_n & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Recalling 2.1 and computing the transformation matrices with the values obtained by the DH convention, the following results are obtained

$$\boldsymbol{p}_6^0 = \begin{pmatrix} L_1c_1c_2 + L_2c_1c_{23} + L_3(c_1(c_{23}c_4s_5 - s_{23}c_5) - s_1s_4s_5) \\ L_1s_1c_2 + L_2s_1c_{23} + L_3(s_1(c_{23}c_4s_5 - s_{23}c_5) + c_1s_4s_5) \\ L_1s_2 + L_2s_{23} + L_3(c_{23}c_5 + s_{23}c_4s_5) \end{pmatrix}$$

$$\boldsymbol{n}_6^0 = \begin{pmatrix} c_1(c_{23}(c_4c_5c_6 - s_4s_6) + s_{23}s_5c_6) - s_1(s_4c_5c_6 + c_4s_6) \\ s_1(c_{23}(c_4c_5c_6 - s_4s_6) + s_{23}s_5c_6) + c_1(s_4c_5c_6 + c_4s_6) \\ -c_{23}s_5c_6 + s_{23}(c_4c_5c_6 - s_4s_6) \end{pmatrix}$$

$$\boldsymbol{s}_6^0 = \begin{pmatrix} c_1(-c_{23}(c_4c_5s_6 + s_4c_6) - s_{23}s_5s_6) - s_1(c_4c_6 - s_4c_5s_6) \\ s_1(-c_{23}(c_4c_5s_6 + s_4c_6) - s_{23}s_5s_6) + c_1(c_4c_6 - s_4c_5s_6) \\ c_{23}s_5s_6 - s_{23}(c_4c_5s_6 + s_4c_6) \end{pmatrix}$$

$$\boldsymbol{a}_6^0 = \begin{pmatrix} c_1(c_{23}c_4s_5 - s_{23}c_5) - s_1s_4s_5 \\ s_1(c_{23}c_4s_5 - s_{23}c_5) + c_1s_4s_5 \\ c_{23}c_5 + s_{23}c_4s_5 \end{pmatrix}$$

where $s_i = sin(\theta_i)$, $c_i = cos(\theta_i)$, $s_{ij} = sin(\theta_i + \theta_j)$ and $c_{ij} = cos(\theta_i + \theta_j)$ [17].

2.2.2 Inverse Kinematics

The inverse kinematics problem consists in the determination of the joint variables corresponding to a given end-effector pose and orientation. The solution of this problem allows to transform the motion specifications assigned to the end-effector in the operational space, into the corresponding joint space motions. Usually it is not guaranteed to have a unique solution to this problem but if the given end-effector pose and orientation belong to the manipulator dexterous workspace the solution is ensured. It is important to notice that for particular end-effector configurations the inverse kinematics problem does not admit solutions.

Starting from 2.1, the goal is to determine the matching joint variable to the pose and orientation of the end-effector. Considering the particular configuration of the e.DO cobot, composed by the combination of an anthropomorphic arm and a spherical wrist. As can be seen in Figure 2.5, the last three consecutive joints axis intercept in a common point and this particular configuration (*spherical wrist*) ensure the presence of a closed form inverse kinematics solution.

The presence of a spherical wrist allows to decouple the inverse kinematics problem into two subproblems since it guarantees the presence of point along the structure, whose position can be expressed both as function of a given end-effector pose and orientation and as a function of a reduced number of joint variables. For the particular configuration of the e.DO cobot, it is convenient to locate such point W at the intersection of the three terminal revolute axis. In fact, once the end-effector pose and orientation are defined in terms of $\boldsymbol{p}_{\mathrm{e}}$ and $\boldsymbol{R}_{\mathrm{e}}$=[$\boldsymbol{n}_{\mathrm{e}}$ $\boldsymbol{s}_{\mathrm{e}}$ $\boldsymbol{a}_{\mathrm{e}}$], the wrist position can be computed as

$$\boldsymbol{p}_W = \boldsymbol{p}_e - d_n \boldsymbol{a}_e \tag{2.2}$$

where d_n is the link between the intersection point of the wrist and the end-effector. The expression 2.2 is a function of the sole joint variables that determine the arm position and if it is a (non-redundant) three-DOF arm, the inverse kinematics can be solved according to these steps:

- Compute the wrist position $\boldsymbol{p}_{\mathrm{W}}(q_1,q_2,q_3)$ as in 2.2;
- Solve the inverse kinematics for (q_1,q_2,q_3);
- Compute $\boldsymbol{R}_3{}^0(q_1,q_2,q_3)$;
- Compute $\boldsymbol{R}_6{}^3(\theta_4,\theta_5,\theta_6)$=$\boldsymbol{R}_3{}^{0\mathrm{T}}\boldsymbol{R}$;
- Solve the inverse kinematics for orientation $(\theta_4,\theta_5,\theta_6)$.

From the direct kinematic computation the following results hold

$$p_{Wx} = c_1(L_2 c_{23} + L_1 c_2)$$

$$p_{Wy} = s_1(L_2 c_{23} + L_1 c_2)$$

$$p_{Wz} = L_2 s_{23} + L_1 s_2$$

Solving the inverse kinematics problem and omitting for brevity all the mathematical computation, that is treated in detail in Appendix B.2, the following results are obtained

- θ_3

$$\theta_3 = atan2(s_3, c_3) = \begin{cases} \theta_{3,I} \in [-\pi, \pi] \\ \theta_{3,II} = -\theta_{3,I} \end{cases} \tag{2.3}$$

where $c_3 = \frac{p_{Wx}^2 + p_{Wy}^2 + p_{Wz}^2 - L_2^2 - L_3^2}{2L_2L_3}$ and $s_3 = \pm\sqrt{1 - c_3^2}$

- θ_2

$$\theta_2 = atan2(s_2, c_2) \tag{2.4}$$

where $c_2 = \frac{\pm\sqrt{p_{Wx}^2+p_{Wy}^2}(L_1+L_2c_3)+p_{Wz}L_2s_3}{L_1^2+L_2^2+2L_1L_1c_3}$ and $s_2 = \frac{p_{Wz}(L_1+L_2c_3)\mp\sqrt{p_{Wx}^2+p_{Wy}^2}L_2s_3}{L_1^2+L_2^2+2L_1L_1c_3}$

So, according to the sign of s_3, four possible solution are available, two for $s_3^+ = +\sqrt{1-c_3^2}$ and two for $s_3^- = -\sqrt{1-c_3^2}$

- θ_1

$$\begin{cases} \theta_{1,I} & = atan2(p_{Wy}, p_{Wx}) \\ \theta_{1,II} & = atan2(-p_{Wy}, -p_{Wx}) \end{cases} \tag{2.5}$$

Since $atan2(-y,-x) = -atan2(y,-x) = \begin{cases} \pi - atan2(y,x) & if\ y \geq 0 \\ -\pi - atan2(y,x) & if\ y < 0 \end{cases}$

the final result is

$$\theta_{1,II} = \begin{cases} atan2(p_{Wy}, p_{Wx}) - \pi & if\ p_{Wy} \geq 0 \\ atan2(p_{Wy}, p_{Wx}) + \pi & if\ p_{Wy} < 0 \end{cases}$$

It is important to notice that for $p_{Wx} = p_{Wy} = 0$ the inverse kinematics problem does not admit a solution and the robot is in a *SINGULARITY* configuration.

In Appendix B.4 are shown all the possible configurations of an anthropomorphic arm compatible with a given wrist position.

For the spherical wrist, the inverse kinematics problem consists in the solution of the Euler angles set ZYZ with respect to Frame 3. So, starting from

$$\boldsymbol{R}_6^3 = \begin{pmatrix} n_{x6}^3 & s_{x6}^3 & a_{x6}^3 \\ n_{y6}^3 & s_{y6}^3 & a_{y6}^3 \\ n_{z6}^3 & s_{z6}^3 & a_{z6}^3 \end{pmatrix}$$

the solution can be directly computed as follow

- θ_4

$$\theta_4 = \begin{cases} atan2(a_{y6}^3, a_{x6}^3) & if\ \theta_5 \in [0, \pi] \\ atan2(-a_{y6}^3, -a_{x6}^3) & if\ \theta_5 \in [-\pi, 0] \end{cases}$$

- θ_5

$$\theta_5 = \begin{cases} atan2(\sqrt{(a_{x6}^3)^2 + (a_{y6}^3)^2}, a_{z6}^3) & if\ \theta_5 \in [0, \pi] \\ atan2(-\sqrt{(a_{x6}^3)^2 + (a_{y6}^3)^2}, a_{z6}^3) & if\ \theta_5 \in [-\pi, 0] \end{cases}$$

- θ_6

$$\theta_5 = \begin{cases} atan2(s^3_{z6}, -n^3_{z6}) & if\ \theta_5\ \epsilon\ [0, \pi] \\ atan2(-s^3_{z6}, n^3_{z6}) & if\ \theta_5\ \epsilon\ [-\pi, 0] \end{cases}$$

An introduction to the ***ZYZ*** *Euler Angles* is shown in Appendix B.3 [17].

2.3 Development Environments

2.3.1 Robot Operating System

Robot Operating System is a trending robot application development platform that provides various features such as hardware abstraction, low-level device control, message passing, distributed computing, code reusing and so on [18]. Similarly to an operating system, the ROS files are organized in memory in a particular fashion. Figure 2.6 shows how files and folders are organized on the disk

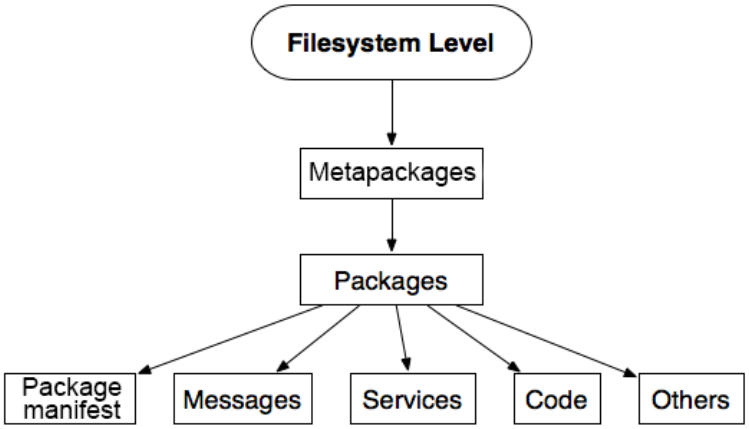

Figure 2.6: ROS File System Level

where each block performs a different task, as shown below:

- **Packages** ⇒ The ROS packages are the most basic unit of the ROS software. A package contains the ROS runtime process (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build item and release item in the ROS in the ROS software;
- **Package manifest** ⇒ The package manifest file is inside a package that contains information about the package, author, license, dependencies, compilation flags, and so on. The *package.xml* file inside the ROS package is the manifest file of the package;

- **Meta packages** ⇒ The term meta package is used for a group of packages for a special purpose. In an older version of ROS such as Electric and Fuerte, it was called stacks, but later it was removed, as simplicity and meta packages come to existence. One of the examples of a meta package is the ROS navigation stack;
- **Meta packages manifest** ⇒ The meta package manifest is similar to the package manifest. Differences are that it might include packages inside it as runtime dependencies and declare an export tag;
- **Messages (.msg)** ⇒The ROS messages are a type of information that is sent from one ROS process to the other. It is possible to define a custom message within the *msg* folder of the package (*Package_name/msg/custom_msg.msg*). The extension of a message file is *.msg*;
- **Services (.srv)** ⇒The ROS service is a kind of request/reply interaction between processes. The reply and request data types can be defined within the *srv* folder of the package (*Package_name/srv/custom_srv.msg*);
- **Repositories** ⇒ Most of the ROS packages are maintained using a **Version Control System** such as Git, subversion, mercurial, and so on. The collection of packages that share a common VCS can be called repositories. The package in the repositories can be released using catkin release automation tool called *bloom*.

For more details about implementation of different blocks, take a look to [19].

2.3.2 Rviz

Rviz is a 3D visualization tool for ROS applications. It allows the visualization of robot models and incoming information acquired from sensors and can display data form video cameras, lasers and 3D and 2D devices, including images and point clouds.

For more details, take a look to [20].

2.3.3 MoveIt

MoveIt is a set of packages and tools for doing mobile manipulation in ROS. The official webpage [21] contains the documentations, the list of robots using *MoveIt*, and various examples to demonstrate pick and place, grasping, simple motion planning using inverse kinematics, and so on. It contains a state of the art software for motion planning, manipulation, 3D perception, kinematics, collision checking, control, and navigation. Apart from the command line interface, MoveIt has some

good GUI to interface a new robot to it. Also, there is the Rviz plugin which enables motion planning from Rviz itself. It allows also to plan the robot motion using the *MoveIt* C++ APIs.

For more details about *MoveIt* architecture, take a look to [19].

2.4 Connection to the Robot

Once the robot has been turned on, two available connection methods are available, to boot the control algorithm and control it:

- Wired connection, setting manually the laptop IP;
- WiFi connection.

More details about the step-by-procedure to follow are shown in Appendix B.5.

Chapter 3

Perception Pipeline

A vision system allows to acquire geometrical and qualitatively information about the environment in which the robot works. To obtain these kind of information the raw image must be processed through different steps, starting from filtering and feature extraction up to extraction of information about scene's objects, according to epipolar geometry algorithms.

A visual system for robotics is characterized by different aspects:

- Type of camera: simple camera or stereo-camera (provides 3D information about the objects in the scene);
- Camera configuration:
  1. if the camera is fixed on a plane, the configuration is said *eye-to-hand*;
  2. if the camera is mounted on the robot end-effector, the configuration is mobile and is said *eye-in-hand*;
  3. it is also possible to have more than one camera with different configurations, this case is said *hybrid configuration*.

In the *eye-to-hand* configuration, the visual system looks at the objects with a pose fixed with respect to the robot base link. This configuration has the benefit of a static field of view that, theoretically, maintain constant the measurements accuracy. On the other hand, when the manipulator is moving within the field of view of the camera, it is possible that it overlaps a portion of the camera's field of view, covering objects in the scene that can be of paramount importance.

In the *eye-in-hand* configuration, the visual system is mounted on the end-effector of the robot, with two available positions: upstream or downstream the wrist. In the first configuration, the camera looks at the end-effector with a favorable position, without occlusion for the scene, meanwhile in the second configuration, the camera moves statically with the end-effector, looking exclusively at the scene.

In both the configurations the field of view of the camera changes radically during motion and this aspect cause a huge gap in the accuracy of the measurements. Anyway, this configuration has the benefit to achieve a constant accuracy in measurements when the end-effector is close to the object of interet within the camera scene and practically, the problem of the overlapping end-effector is practically solved.

In the *hybrid configuration* it is possible to achieve all the benefits of the two configurations, solving the problem of the overlapping end-effector at the same time.

The aim of this section is to identify specific objects (to realize the demonstrator for the Aloha Smart Industry applications use case) within the Field of View of the camera, in the particular case all the cylinders that are available in the scene. To achieve this target, the raw image coming from the camera is processed through some modules of the PCL library, as will be described in details in the next paragraphs.

3.1 Intel RealSense D435i

The configuration chosen for this thesis is the eye-to-hand one with the camera fixed on the same plane of the robot with the following transformation characteristics with respect to *edo_base_link*:

- $[x,\ y,\ z] = [1.5,\ 0,\ 0.23]$ (meters)
- $[r,\ p,\ y] = [0,\ 0.349,\ 3.14]$ (rad)

The device selected as camera sensor is the Intel RealSense D435i, having the following structure an characteristics:

- Features:
  - Use environment: Indoor/outdoor;
  - Image Sensor Technology: Global Shutter, 3μm x 3μm pixel size;
  - Maximum Range: Approximately 10 meters. Accuracy varies depending on calibration, scene and lighting condition.
- Depth:
  - Depth Technology: Active IR Stereo;
  - Depth Field of View (FOV): 87°±3° x 58°±1° x 95°±3°;
  - Minimum Depth Distance (Min-Z): 0.105 m;

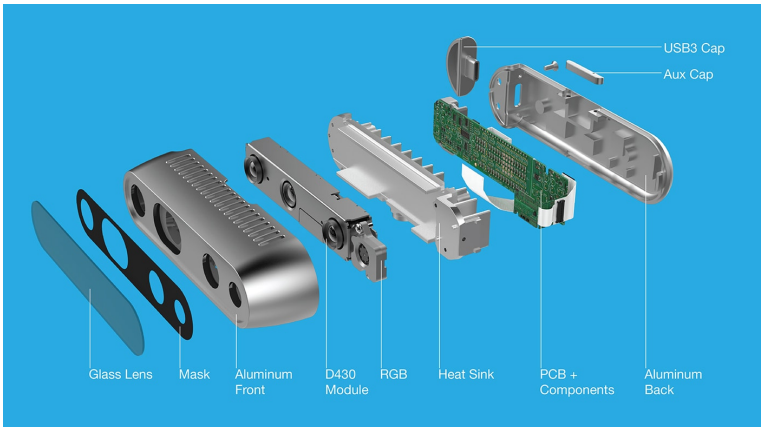

Figure 3.1: Internal structure of the Intel RealSense D435i camera

  - Depth Output Resolution & Frame Rate: Up to 1280*720 active stereo depth resolution. Up to 90 FPS.

- RGB:
  - RGB Sensor Resolution & Frame Rate: 1920*1080;
  - RGB Frame Rate: 30 FPS;
  - RGB Sensor FOV (H x V x D): 69.4° x 42.5° x 77°±3°.
- Major Components:
  - Camera Module: Intel RealSense Module D430 + RGB Camera;
  - Vision Processor Board: Intel RealSense Vision Processors D4.
- Physical:
  - Form Factor: Camera Peripheral;
  - Connectors: USB-C* 3.1 Gen 1*;
  - Length x Depth x Height: 90mm x 25mm x 25mm;
  - Mounting Mechanism: One 1/4-20 UNC thread mounting point, two M3 thread mounting points.

3.2 PCL Library

The Point Cloud Library (PCL) is a large scale, open project for point cloud processing. The PCL framework contains numerous state-of-the art algorithms, including filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation. These algorithms can be used, as examples, to filter outliers from noisy data, stitch 3D point cloud together, segment relevant parts of a scene, extract keypoints and compute descriptors to recognize objects in the world based on their geometric appearance, create surfaces from point clouds and visualize them.

PCL is cross-platform, and has been successfully compiled and deployed on Linux, MacOS, Windows, and Android. To simplify development, PCL is split into a series of smaller code libraries, that can be compiled separately. This modularity is important for distributing PCL on platforms with reduced computational or size constraints.

Furthermore it is released under the terms of the BSD license and is open source software. **It is free for commercial and research use** [22].

The whole chapter is dedicated to the explanation of software project to achieve the target described previously, considering as camera reference frame the triad in Figure 3.2. It is important to notice that this triad is used within the software to make computations of objects characteristics and to spawn them in the planning scene.

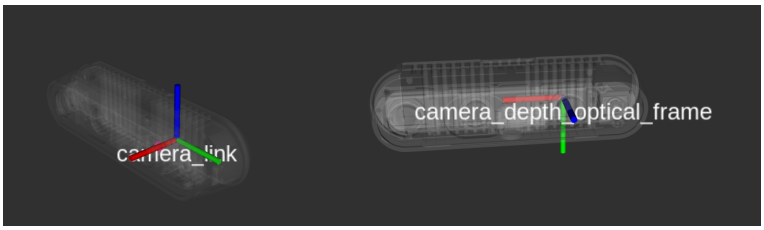

Figure 3.2: Camera reference triad chosen for the software development (right triad)

A brief introduction is shown here to highlight the main steps performed by the software:

1. Subscribe to the topic containing data coming from the stereo camera and call the CloudCB (Cloud Call Back) function;
2. Convert the sensor_msgs data to pcl::PointCloud<pcl::PointXYZRGB> type;
3. Add a pass-through filter to reduce the region of interest within the desired ROI;

4. Apply a statistical outlier filter to reduce noise of the point cloud;
5. Remove the planar surface on which items lie;
6. Extract clusters from the Point cloud and then, for each cluster:
   (a) Smooth cluster through polynomial reconstruction;
   (b) Compute normals;
   (c) Perform 3D object recognition through hypotheses verification;
   (d) If hypotheses are verified:
      - Extract normals;
      - Estimate the cylinder parameters;
      - Extract pose and height of the cylinder;
      - Estimate the color of the cylinder according to ***HSV*** colorspace;
      - Add the cylinder to the planning scene;
7. Skipping only the first acquisition, update the planning scene;
8. Wait an input from keyboard (or a serial input from SensorTile) to make a new acquisition.

3.2.1 Filter in ROI

The first step consists in the conversion of the sensor_msgs raw data coming from the camera, to a point cloud object containing information about points coordinates and color (in terms of RGB standard). Once all the data have been converted, the Point Cloud is filtered to contains only data within a certain Region Of Interest, fixed along the Z axis of the triad in Figure 3.2, between these minimum and maximum values: [0.3m, 1.1m]. After the filtering process, the result can be seen in Figure 3.3.

3.2.2 Statistical Outlier Removal Filter

The aim of this step is to solve some irregularities by performing a statistical analysis on each point's neighbor, and trimming those which do not meet a certain criteria. This filter is based on the compensation of the distribution of points to neighbor distances in the input dataset.

For each point, the mean distance from the point to all its neighbors is computed. Assuming that the resultant distribution is Gaussian with a mean and a standard deviation, all points whose mean distances are outside an interval defined by the global distances mean and standar deviation, can be considered as outlier and trimmed from the dataset, as shown in Figure 3.4.

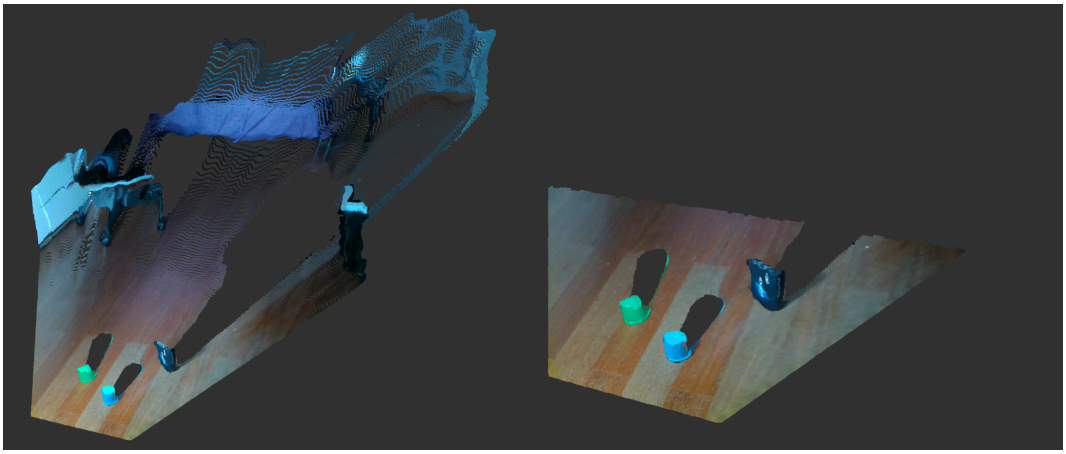

Figure 3.3: Point Cloud before (left) and after (right) filtering

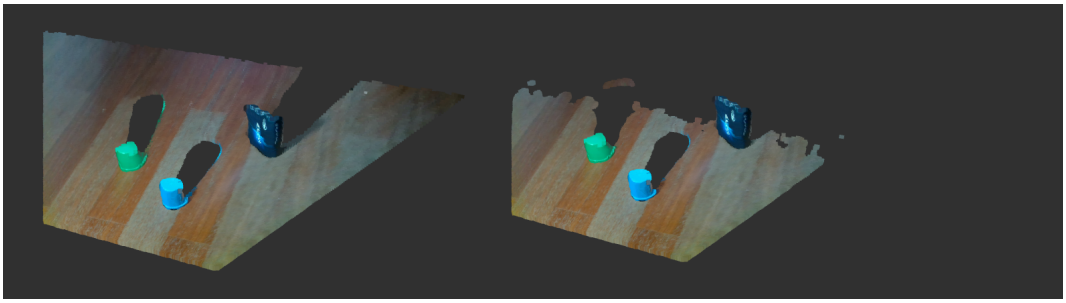

Figure 3.4: Point Cloud before (left) and after (right) Statistical Outlier Removal

3.2.3 Removal of Planar Surface

Once the Point Cloud has been confined within the ROI, all the point indices that correspond to a planar surface are stored to clear the Point Cloud. In this way the Point Cloud will contain only information relative to the objects available in the captured scene, as can be seen in Figure 3.5. This step is based on the RANSAC method, which is an iterative method to estimate a model's parameter from a starting set of data containing *outlier*. It is a non deterministic algorithm that produces a result with a certain probability, that increase with the maximum iterations allowed.

A basic assumption is that the data consists of *inliers*, that are data whose distribution can be explained by some set of model parameters, and *outliers* which are data that do not fit the model (data coming from extreme values of the noise or from erroneous measurements or incorrect hypotheses about the interpretation

of data). RANSAC assumes that, given a set of inliers, typically small, there exists a procedure that can estimate the parameters of a model that optimally explains or fits these data [23].

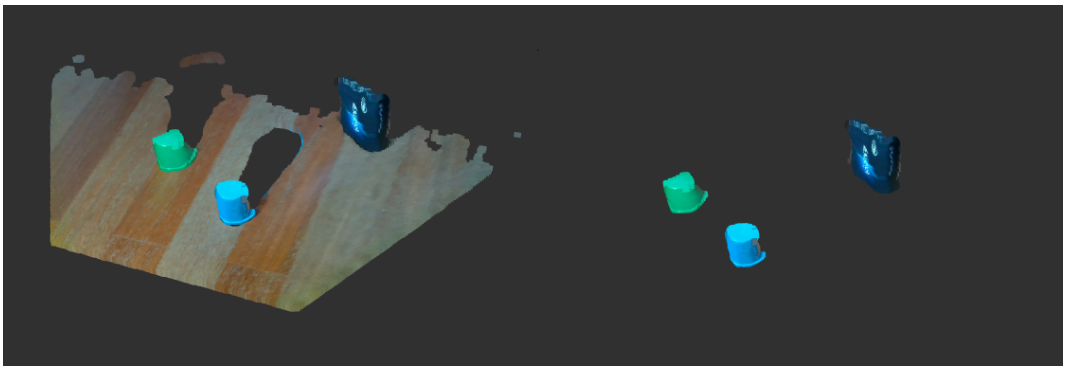

Figure 3.5: Point Cloud before (left) and after (right) planar surface removal

3.2.4 Euclidean Clusters Extraction

This method is based on the Euclidean Cluster Extracion. A clustering method needs to divide an unorganized point cloud model P into smaller parts so that the overall processing time for P is significantly reduced. A simple data clustering approach in an Euclidean sense can be implemented by making use of a 3D grid subdivision of the space using fixed width boxes, or more generally, an octree data structure. This particular representation is very fast to build and is useful for situations where either a volumetric representation of the occupied space is needed, or the data in each resultant 3D box (or octree leaf) can be approximated with a different structure. In a more general sense however, it is possible make use of nearest neighbors and implement a clustering technique that is essentially similar to a flood fill algorithm. The aim is to find and segment the individual object point clusters lying on the plane. Assuming to use a Kd-tree structure for finding the nearest neighbors, the algorithmic steps for that would be:

1. create a Kd-tree [24] representation for the input point cloud dataset P;
2. For every point $\boldsymbol{p}_i \in P$, perform the following steps:
   (a) Set up an empty list of clusters C, and a queue of the points that need to be checked Q;
   (b) Add $\boldsymbol{p}_i$ to the current queue Q;
   (c) For every point $\boldsymbol{p}_i \in Q$ do:

i. Search for the set P_k^i of point neighbors of $\boldsymbol{p}_i$ in a sphere with radius r $< d_{th}$;
ii. For every neighbor $\boldsymbol{p}_i^k \in P_i^k$, check if the point has already been processed, and if not add it to Q;
iii. When the list of all points in Q has been processed, add Q to the list of clusters C, and reset Q to an empty list

3. The algorithm terminates when all points $\boldsymbol{p}_i \in P$ have been processed and are now part of the list of point clusters C.

It is important to set correctly the parameters and variables for extraction and in particular it is of paramount importance the parameter *setClusterTolerance()*, because if it is chosen too small, it can happen that an object is interpreted as multiple clusters and if too high, it can happen that multiple objects are seen as a single cluster.

Smooth Clusters through Polynomial Reconstruction

This step is introduce to smooth and resample noisy data, coming from the device. In this application it is necessary, since accordingly to the camera settings, the raw point clouds become more noisy with the temperature of the device, preventing the correct execution of all the next steps of 3D recognition and object parametrization.

This algorithm is based on a Moving Least Square (MLS) surface reconstruction and is useful to remove some data irregularities, caused by small distance measurements errors, that can be very hard to remove through statistical analysis. On the other hand, this method guarantees the creation of complete models, taking into account the presence of glossy surfaces as well as occlusions.

Practically this method performs a resampling algorithm, which attempts to recreate the missing parts of the surface by higher order polynomial interpolations between the surrounding data points. The surface defined by a local neighborhood of points p_1, p_2, ..., p_k and point q is approximated by a bivariate polynomial height function, defined on a robustly computed reference plane.

According to the scene presented in Figure 3.5, the clusters extracted from the Point Cloud are shown in Figure 3.6.

Computation and Extraction of Point Normals

Surface normals are important properties of a geometric surface, and are heavily used in many areas such as computer graphics applications, to apply the correct light sources that generate shadings and other visual effects. It is typically trivial to determine the direction of the normal at a certain point on the surface as the vector perpendicular to the surface in that point. Anyway, considering that a point

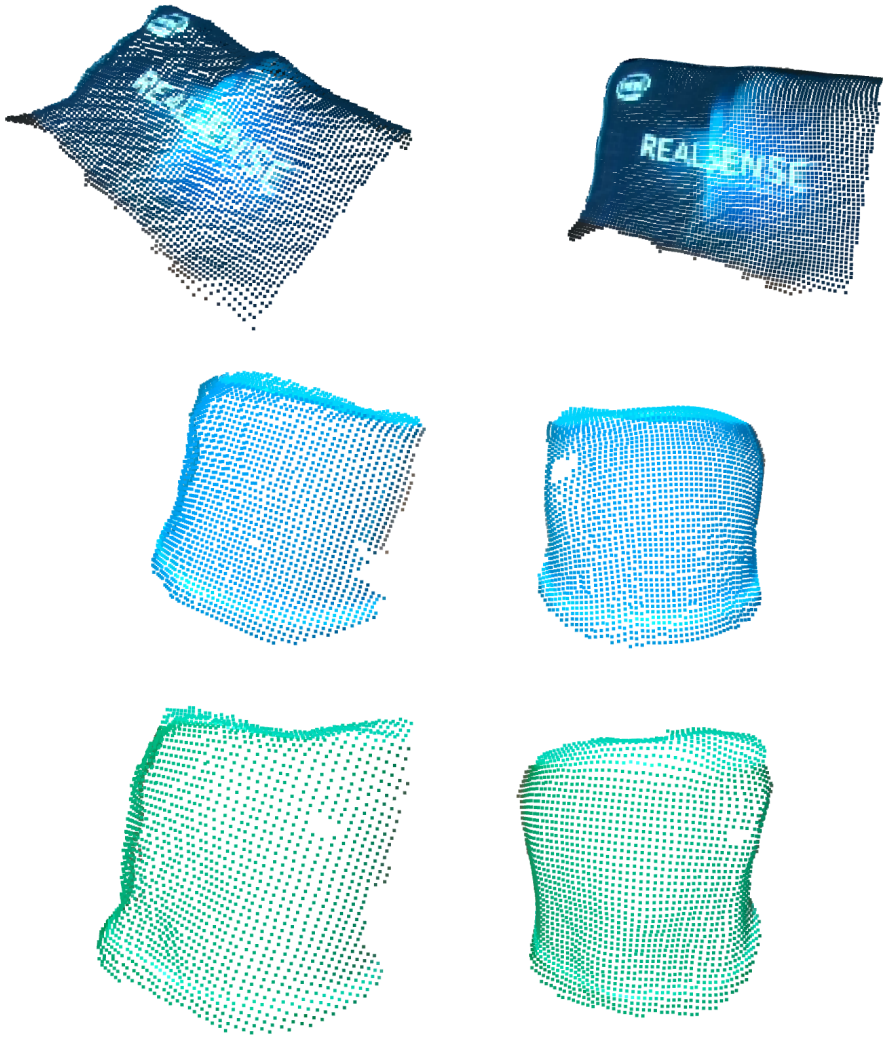

Figure 3.6: Cloud clusters before (left) and after (right) polynomial reconstruction

cloud dataset represents a set of point samples on the real surface, two possibilities are available:

- Obtain the underlying surface from the acquired point cloud dataset, using surface meshing techniques, and then compute the surface normals from the mesh;
- Use approximations to infer the surface normals from the point cloud dataset

directly.

In this case, the latter solution has been adopted, that given a point cloud dataset, directly compute the surface normals at each point in the cloud.

A surface normal at a point needs to be estimated from the surrounding point neighborhood support of the point (the so-called ***k-neighborhood***). The specifics of the nearest-neighbor estimation problem raise the question of the *right scale factor*, that consists in the identification of the correct parameter **k** (given by pcl::Feature::setKSearch) or **r** (given by pcl::Feature::setRadiusSearch) that should be used to determine the set of nearest neighbors of a point.

This issue is of extreme importance and constitutes a limiting factor in the automatic estimation of a point feature representation. In Figure 3.7 the differences between selecting a small or large scale can be seen.

It is important to notice that the normal extraction step is performed after the 3D object recognition, to avoid useless computations if no match with the cluster is found.

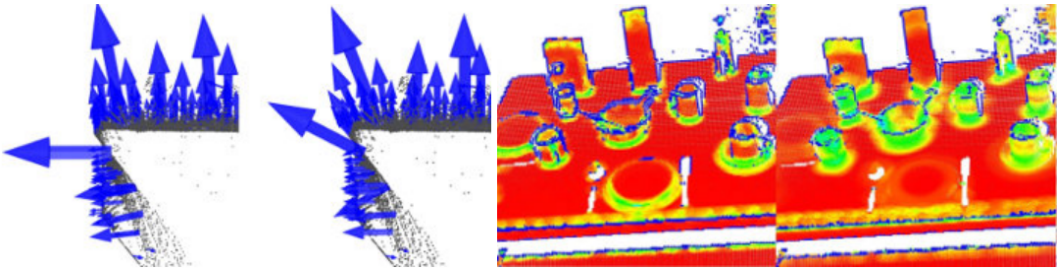

Figure 3.7: Normals computation with small scale (left) and large scale (right)

The left part of the figure depicts a well chosen scale factor, with estimated surface normals approximately perpendicular to the two planar surfaces. Anyway if the scale factor is too high (right part of the figure), the set of neighbors is larger covering points from adjacent surfaces and the estimated point feature representation get distorted, with rotated surface normals at the edges of the two planar surfaces, smeared edges and suppressed fine details.

So, to set this value, it is sufficient to assume that the scale factor for the determination of a point's neighborhood has to be selected based on the level of details that is required by the application. Practically, if the greatest part of the details in the image is necessary to go on in the image processing, the scale factor needs to be small enough to capture those details, and large otherwise.

Once all the normals have been computed and their relative indices have been extracted, the normals that correspond to the plane on which cylinder lies, will be used to extract the object form the cluster.

For this step and for all the normal estimations steps, the PCL additional OMP implementation has been used, which uses multi-core/multi-threaded paradigms to speedup computation.

Hypotheses Verification for 3D Object Recognition

1. Prerequisites:
   - Within the folder "*src/edo_perception/add/models/*" add all the models relative to the objects of interest, in the form "*cloud_cluster_model_n.pcd*" and modify the models number within the file "*models_cnt.txt*";
   - Start the node from the workspace directory, to avoid path mismatches.
2. Step-by-step description:
   - Load cluster model and normal computation ⇒ Load the "*.pcd*" file relative to the model through the PCL method. At this point if the file is not correctly loaded for any reason, the execution of the 3D object recognition is interrupted and the cylinder parametrization is skipped. For the normal estimation the procedure is the same as ***Computation and Extraction of Point Normals*** already discribed;
   - Downsample both model and cluster to to find a small number of keypoints, which will then be associated to a 3D descriptor, in order to perform keypoint matching and determine point-to-point correspondances;
   - Associate a 3D descriptor to each model and scene keypoint through SHOT descriptors. They estimate the *Signature of Histogram of OrienTations* descriptors for a given point cloud dataset containing points and normals;
   - Determine point-to-point correspondances between model and scene descriptors. To do so, a KdTreeFLANN (a generic type of 3D spatial locator using KdTree structure that makes use of Fast Library for Approximate Nearest Neighbor) is use, whose input cloud has been set to the cloud containing model descriptors. For each descriptor associated to a scene keypoint, it efficiently finds the most similar model descriptor based on the Euclidean distance, and it adds this pair to a correspondences vector only if the two descriptors are similar enough (i.e. their squared distance is less than a threshold set to 0.25);
   - Perform a clustering algorithm on the previously found correspondances, based on an Hough Voting process. The Hough Voting algorithm requires to associate a Local Reference Frame (LRF) to each keypoint belonging to the clouds which are passed as arguments. To do so, the set of LRFs is computed using BOARDLocalReferenceFrame estimator (BOrder Aware

Repeatable Directions algorithm for local reference frame estimation), before calling the clustering algorithm;

- Apply some ICP (Iterative Closest Point) iterations, to improve the "*coarse*" transformation associated to each object hypothesis. To do so, an "*instances*" list is created, whose aim is to store the "*coarse*" transformations. Then ICP runs over the instances with respect to the scene, in order to obtain the vector of "*registered_instances*". This step provides information about the status of each instance with respect to the model, practically identifying if the cluster is and the model are aligned or not;
- Perform the Global Hypotheses Verification step. It takes as input the list of *registered_instances* and the real scene (the cluster) to perform a verification step between them to obtain a *hypotheses_mask*, that practically is a bool array with a ***TRUE*** value for the *i-th element* if "*registered_instances[i]*" is a verified hypothesis, and a ***FALSE*** value if it has been classified as a false positive (and it must be rejected);
- (Optional) Visualization step. It performs the results visualization according to the following rules:
  - Display good keypoints in scene and model with a *styleViolet* color;
  - iterate on each instance to:
    - Display "*instance[i]*" with a *styleRed* color;
    - Display "*registered_instance[i]*" in *styleGreen* color if it is verified ("*hypotheses_mask[i] = TRUE*);
    - Display "*registered_instance[i]*" in *styleCyan* color if it is not verified ("*hypotheses_mask[i] = FALSE*).

  This last step can be omitted from the computation simply toggling the "*visualization_flag*" to *FALSE*, within the file "*3DObjRecognition.cpp*".

All the just described steps are performed according to the following rules:

- If no instances are found, repeat each step until at least a *GOOD* instance is found or all the models have been checked;
- If a *GOOD* instance is found, stop the 3D object recognition algorithm and return the integer **1** to go on with the computation of the object parameters;
- If *BAD* instances are found, iterate over all the models and if no match is found return the integer **0** to skip the parameter's computation for the actual object.

Here the models used for the use case are reported in Figure 3.8.

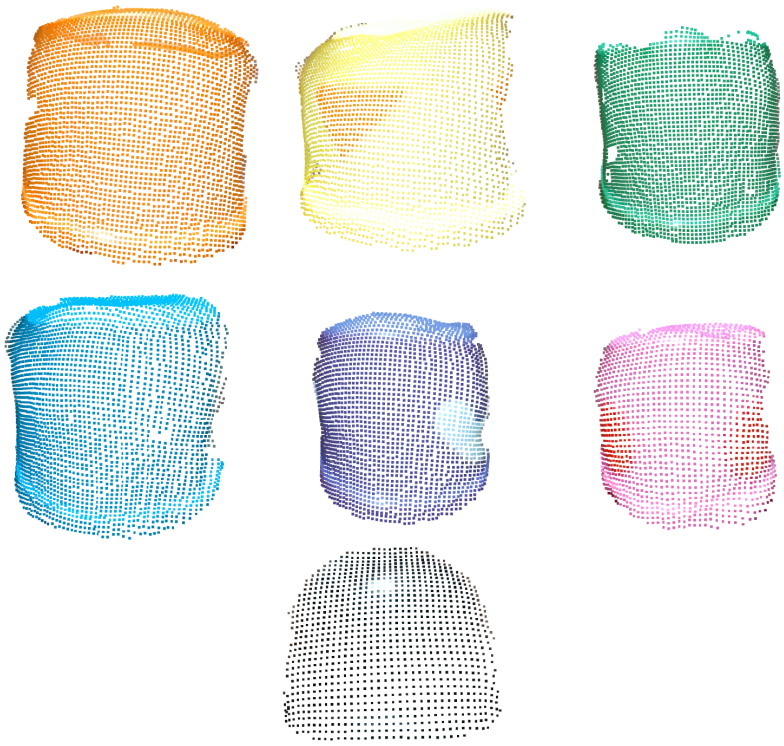

Figure 3.8: Point Cloud models

Object's Parameters Estimation

Through RANSAC robust estimator the cluster is analyzed to extract cylinder coefficients. In this step the radius of interest for the cylindrical object is set within 0 cm up to 10 cm. This value has been set taking into account the maximum opening of the robot gripper an adding some threshold to identify all the cylinders of interest. In fact, considering that the maximum opening value is about 8 cm, the only objects of interest would be all the cylinders having a radius at most equal to $\frac{maxOpening}{2} - thr$, where thr has been experimentally set to 0.5 cm. Then the cylinder inliers and coefficients are computed and the inliers are extracted from the input cloud.

At this point a check on the cluster is done and if it is not empty, the information about the cylinder are stored into a structure with four fields, whose aim is to hold all the parameters that are necessary to define the collision object. The structure fields are:

- ***double*** *radius* ⇒ Field containing the radius of the cylinder;
- ***double*** *direction_vect*[3] ⇒ Field containing the direction vector towards the z-axis of the cylinder;
- ***double*** *center_pt*[3] ⇒ Field containing the center point of the cylinder;
- ***double*** *height* ⇒ Field containing the height of the cylinder.

Anyway not all the parameters are defined and the available data are not enough to define a collision object, since the actual pose and height are still unknown. So to compute the last two unknowns, standard geometry is used, as reported in the next paragraph.

Pose and Height Extraction

Consider a point within a point cloud and imagine that this point is formed on an XY plane, where the perpendicular distance from the plane to the camera is Z.

The perpendicular drawn from the camera to the plane XY hits at center of it, obtaining the x and y coordinates of the point on the plane. Considering X as the horizontal axis, Y as the vertical axis, as shown in Figure 3.2 and C as the center point of the plane, which is Z meters away from the center of the camera, to compute the geometric distance from a generic point to the camera, it is sufficient to compute the hypotenuse that connect these two points. Since this distance is not of interest in the definition of the cylinder parameters (it is basically the Z coordinate of the plane on which the point lies), a loop over the whole point cloud is introduced to find the highest and the lowest point of the cylinder thorough the following steps:

1. Define a variable for the maximum and minimum angles and set them respectively to 0 and ∞;
2. Define two vectors that will contain the coordinates of the highest and lowest points;
3. For each point in the point cloud do:
   (a) If the angle between axis Y and the point is lower than the minimum value of the angle, update the minimum angle and save the coordinates of the new lowest point;
   (b) If the angle between axis Y and the point is greater than the maximum value of the angle, update the maximum angle and save the coordinates of the new highest point;

4. Once the highest and lowest points have been found, compute the center point of the object as:
   - $c_x = \frac{H_x + l_x}{2}$;
   - $c_y = \frac{H_y + l_y}{2}$;
   - $c_z = \frac{H_z + l_z}{2}$.
5. Finally compute the height of the cylinder considering that highest and lowest points can be identified in two different parallel planes *(X, Y)* and/or with different x coordinate. In fact it would be appropriate to compute the height of the cylinder as the geometric distance between these two points:

$$h = \sqrt{(l_x - H_x)^2 + (l_y - H_y)^2 + (l_z - H_z)^2}$$

A graphical representation, considering two points of the point cloud can be seen in Figure 3.9.

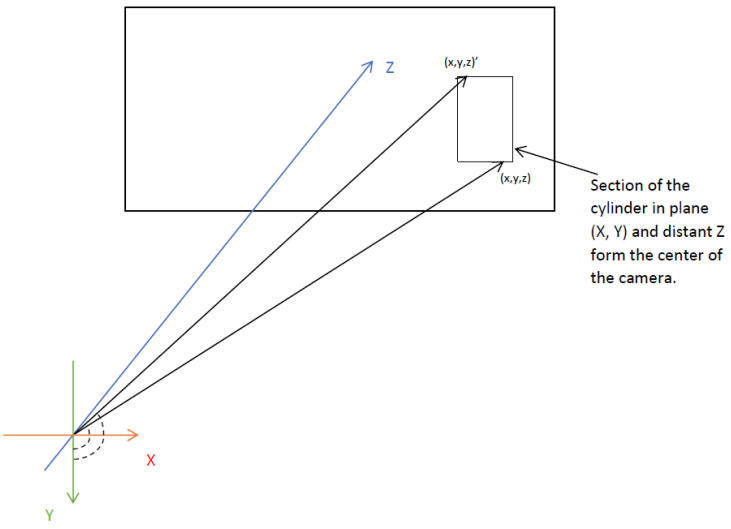

Figure 3.9: Graphical representation of the center point and height estimation

HSV Color Estimation

Another important characteristic for the demonstrator is to differentiate the available objects on the plane by colors. To achieve this goal a function has been generated, whose aim is to convert the RGB color channels to the HSV colorspace

representation. This conversion is necessary since reduces to the minimum the possibility to obtain wrong color estimation from clusters.

The *HSV* colorspace representation is based on the following three parameters:

- *HUE*: Identifies the monochromatic color of the spectrum. It is defined by a value within 0° and 360°, measured as the angle around the vertical axis of the color cone in Figure 3.10;
- *SATURATION*: Identifies the brilliance and intensity of a color;
- *VALUE*: Identifies the lightness or darkness of a color, in term of light reflected.

A graphical representation of the colorspace cone is shown in Figure 3.10

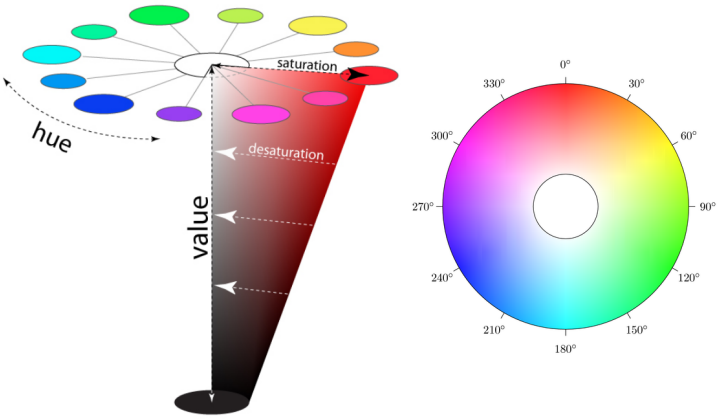

Figure 3.10: HSV cone and parameters definition

Here, for each point in the cluster, the following steps are performed:

1. According to the PCL documentation [25], the point RGB value is unpacked into the three different channels R, G and B;
2. The color channels are normalized as follow (notice that after unpacking the three channels are of type unsigned char so to obtain their numerical values casting to integer is necessary):
   - double $r' = \frac{int(r)}{255}$;
   - double $g' = \frac{int(g)}{255}$;
   - double $b' = \frac{int(b)}{255}$.

Color	**Hue**	**Saturation**	**Value**
Black	$0° < H < 360°$	$0 < S < 100$	$V < 10$
Gray	$0° < H < 360°$	$S < 15$	$10 < V < 65$
White	$0° < H < 360°$	$S < 15$	$V > 65$
Red	$H < 11°$, $H > 351°$	$S > 70$	$V > 10$
Pink	$H < 11°$, $H > 351°$ $310° < H < 351°$	$S < 70$ $S > 15$	$V > 10$ $V > 10$
Orange	$11° < H < 45°$	$S > 15$	$V > 75$
Brown	$11° < H < 45°$	$S > 15$	$10 < V < 75$
Yellow	$45° < H < 64°$	$S > 15$	$V > 10$
Green	$64° < H < 150°$	$S > 15$	$V > 10$
Blue-Green	$150° < H < 180°$	$S > 15$	$V > 10$
Blue	$180° < H < 255°$	$S > 15$	$V > 10$
Purple	$255° < H < 310°$	$S > 50$	$V > 10$
Magenta	$255° < H < 310°$	$15 < S < 50$	$V > 10$

3. After normalization, the conversion from **RGB** to **HSV** is performed:

$$C_{max} = max\{r', g', b'\}$$

$$C_{min} = min\{r', g', b'\}$$

$$\Delta = C_{\{}max\} - C_{\{}min\}$$

$$HUE = \begin{cases} 0° & if \quad \Delta = 0 \\ 60°(\frac{g'-b'}{\Delta}) & if \quad C_{max} = r' \\ 60°(\frac{b'-r'}{\Delta} + 2) & if \quad C_{max} = g' \\ 60°(\frac{r'-g'}{\Delta} + 4) & if \quad C_{max} = b' \end{cases}$$

$$SATURATION = \begin{cases} 0 & if \quad C_{max} = 0 \\ \frac{\Delta}{C_{max}} & if \quad C_{max} \neq 0 \end{cases}$$

$$VALUE = C_{max}$$

   Notice that SATURATION and VALUES will be used in percentage values, so all the results are multiplied by the scale factor **100**.

4. Since no particular requests has been requested on color identification, the code has been structured to identify colors according the following assumptions:

5. Once the color of the point has been identified, a vector containing all the colors occurrences is generated and the occurrences are updated at each cycle, until all the points in the cluster are processed.

Once all the points have been processed, the occurrences vector is scanned to check which is the color with the greatest number of occurrences and the object color is approximated as the color with the maximum occurrences in the vector.

To identify it, the list is scanned and the index of the vector corresponding to the maximum occurrences is stored, once the index is know it is easy to identify the color, considering that the occurrences vector is generated always in the same order, reported below:

$$\begin{bmatrix} BLACK\_OCCURRENCES \\ GREY\_OCCURRENCES \\ WHITE\_OCCURRENCES \\ RED\_OCCURRENCES \\ PINK\_OCCURRENCES \\ ORANGE\_OCCURRENCES \\ BROWN\_OCCURRENCES \\ YELLOW\_OCCURRENCES \\ GREEN\_OCCURRENCES \\ BLUE-GREEN\_OCCURRENCES \\ BLUE\_OCCURRENCES \\ PURPLE\_OCCURRENCES \\ MAGENTA\_OCCURRENCES \end{bmatrix}$$

Adding Object to the Planning Scene

At this point of the computation all the parameters of the object have been correctly computed and are available to generate and add the collision object to the planning scene.

A moveit_msgs::CollisionObject object is created to hold the characteristics of the item that has to be added to the planning scene, and the pipeline proceeds according to two main steps:

1. Only for the first run of the script, the point cloud is processed according to the procedure described above and the collision objects are added to the planning scene as *CYLINDER* primitives. In this step all the cylinders parameters are stored in a vector of double in the order [center_pt_x; center_pt_y; center_pt_z; height; radius] and all the names relative to these cylinders are stored in a vector of strings;

2. For all the next cycles, the code becomes a little bit complex, including a control sequence on the objects already available in the scene. In particular:

   - To check if an incoming object is the same of a previous computation, considering that the objects order may be different in each computation, the procedure is the following.

     For each element in *previous_computation* vector, do:

(a) Check if the height of the incoming object is equal (within a certain tolerance) to the height of one of the previous objects. If this condition is respected, check the radius of this object;

(b) Check if the radius of the incoming object is equal (within a certain tolerance) to the radius of the object highlighted in the first step. If also this condition is respected, the incoming object is probably one of the previous computation;

(c) At this point a check on the center point is performed to check if the object position has undergone changes or not. If it has been moved (within a certain tolerance), it is removed and updated, otherwise it is left unchanged in the planning scene.

- If the incoming object does not match one of the previous, a new object is added to the planning scene with a new name.

For this sequence, the main design parameter was the tolerance used for the control. In fact, considering that the uncertainty of the end-effector pose can assume significant values according to the position that it has to reach within the workspace, the tolerance value has been set to half of a centimeter, in order to have a simulation setup as realistic as possible.

In Appendix C.2 is shown the procedure to estimate the pose of the collision object according to the *quaternion* representation.

3.2.5 Update the Planning Scene

Once all the new objects have been added and all the previous objects have been updated, the last control sequence of the script perform the update of the planning scene. This section checks the objects available in the planning scene and, through a control on the name vectors, check which are the elements that are still present in the scene and which not. At this point, the objects that are no longer available, are removed from the planning scene.

Code snippets of the last two steps are shown in Appendix C.

3.3 Obtained Output

Here a behavioural example is shown:

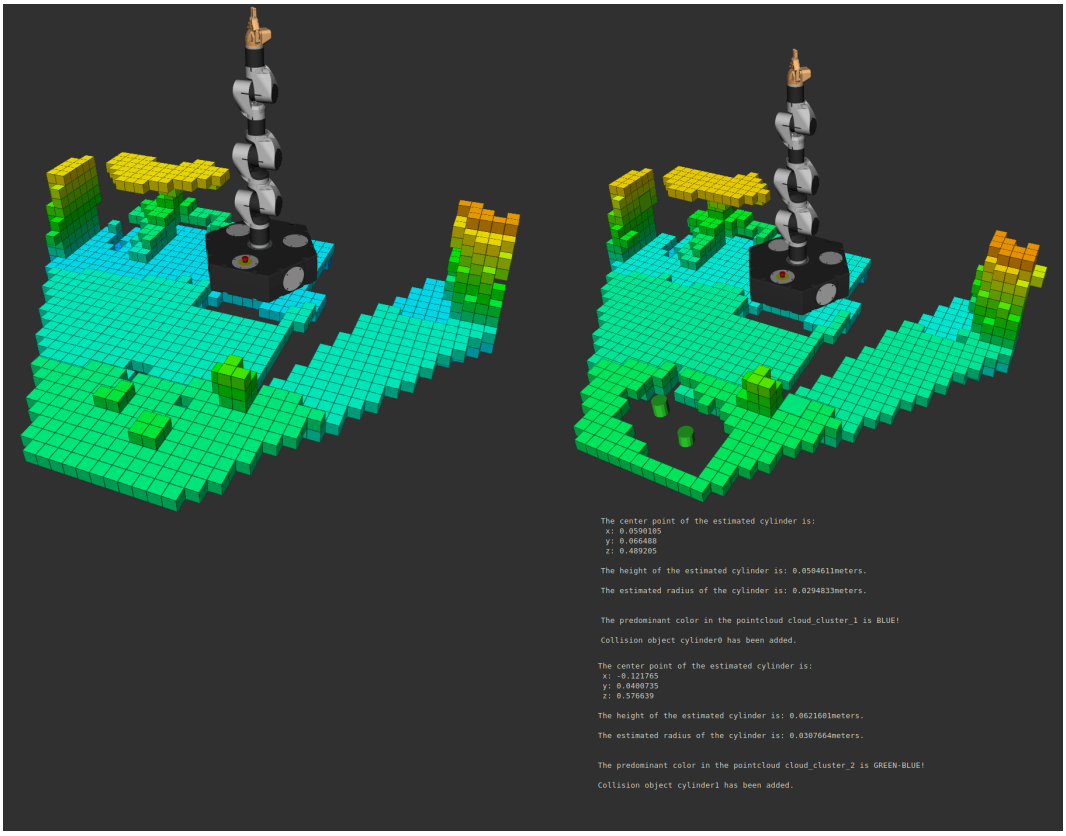

Figure 3.11: Simulation configuration before (left) and after processing(right)

As can be seen from the computational results, each cluster extracted from the pointcloud is compared with respect to each available model, generating a list of instances that can correspond to a correct or wrong match. In particular, if there is no match with the current model the process iterates over all the models as shown from Figure 3.12 to Figure 3.18. On the other hand if a good match is found for the instance, as can be seen in Figure 3.19, the process stops, starting to perform the computation of the cylinder parameters within the *good match* cluster. It is important to notice that the 3D object recognition algorithm perform just a match between model and cluster, meanwhile the parameters computation is computed through the **SAC_CYLINDER** segmentor, which allows the computation of very good results also with a limited number of valid data and a reduced number of iterations.

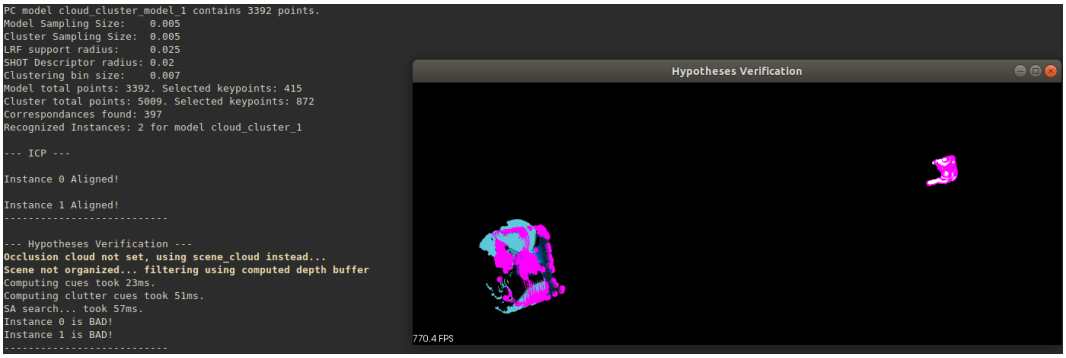

Figure 3.12: 3D Object Recognition wrt Model1 for cluster 1

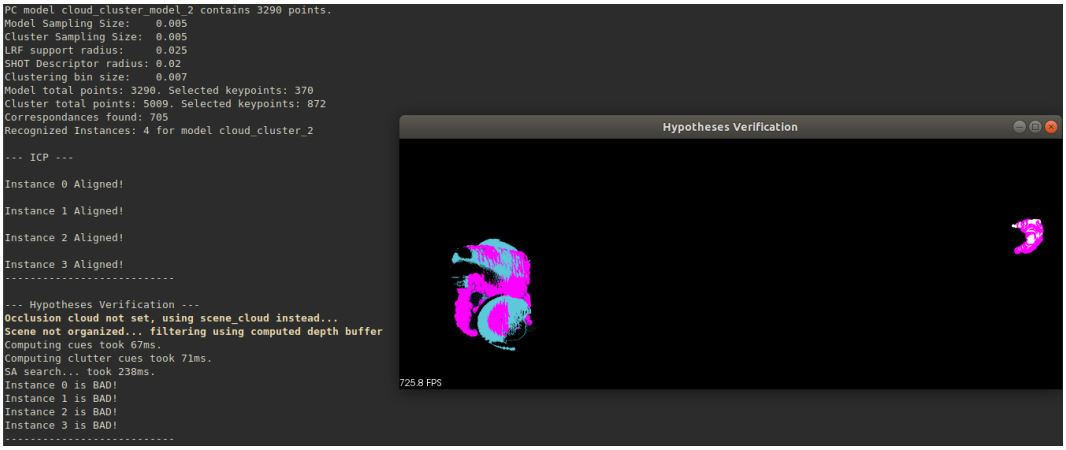

Figure 3.13: 3D Object Recognition wrt Model2 for cluster 1

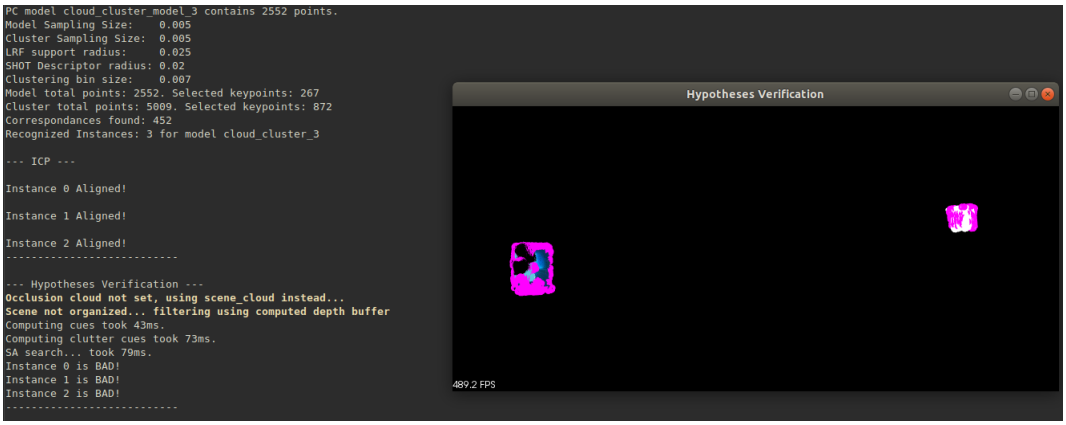

Figure 3.14: 3D Object Recognition wrt Model3 for cluster 1

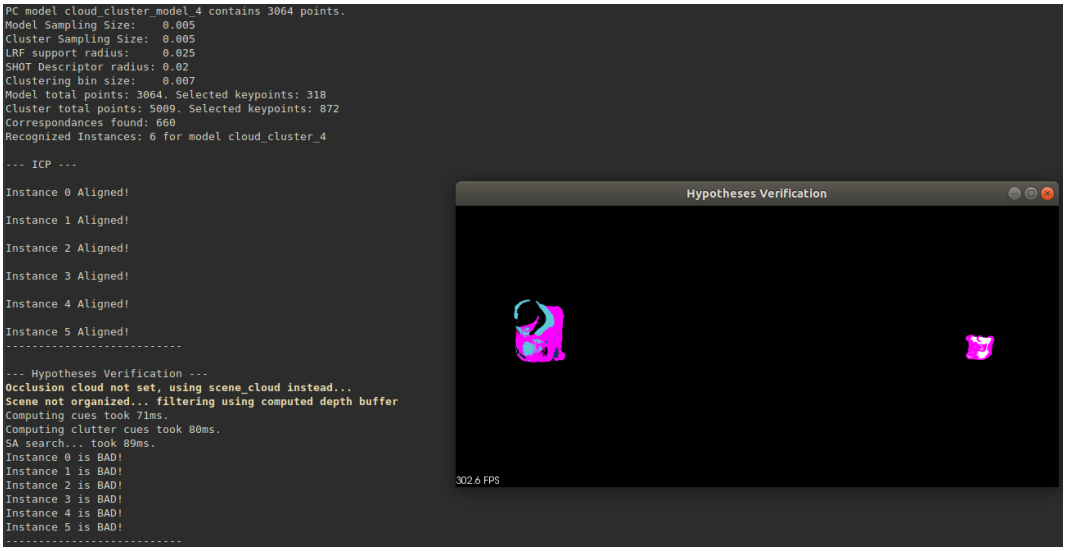

Figure 3.15: 3D Object Recognition wrt Model4 for cluster 1

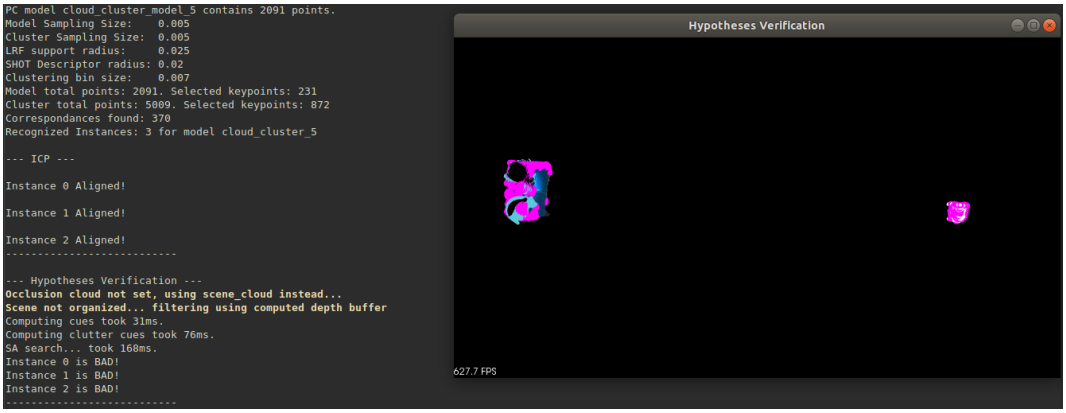

Figure 3.16: 3D Object Recognition wrt Model5 for cluster 1

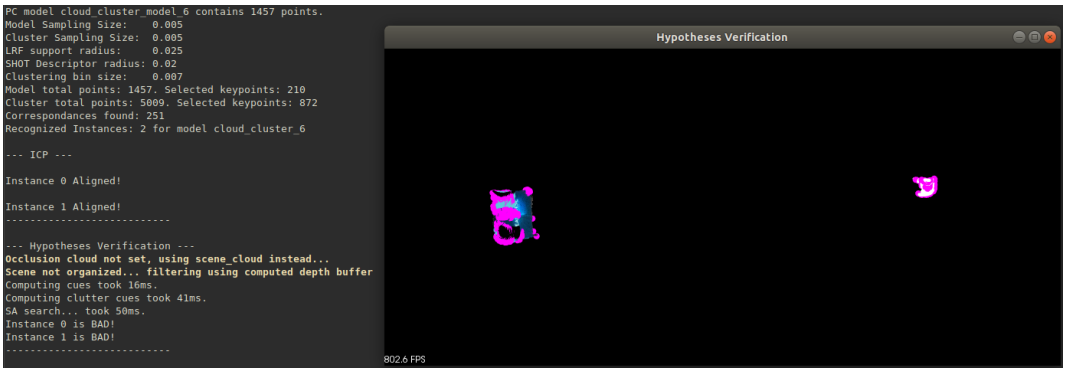

Figure 3.17: 3D Object Recognition wrt Model6 for cluster 1

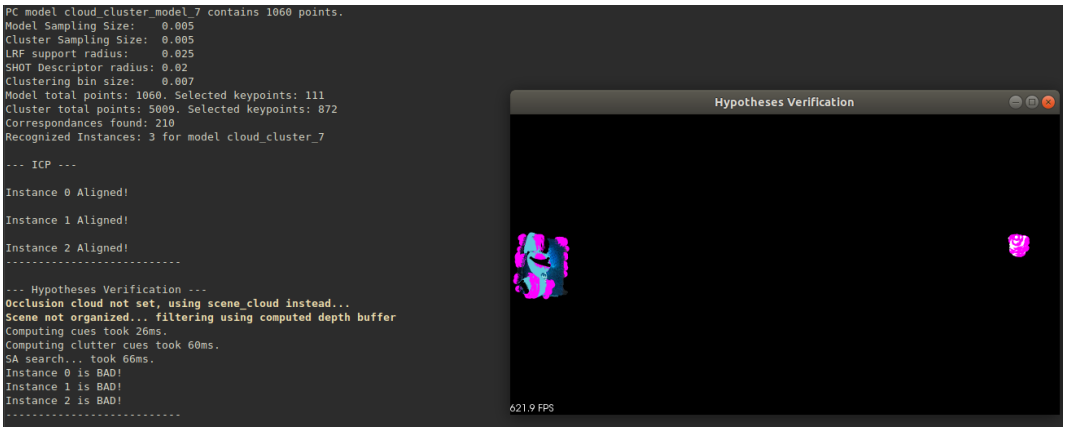

Figure 3.18: 3D Object Recognition wrt Model7 for cluster 1

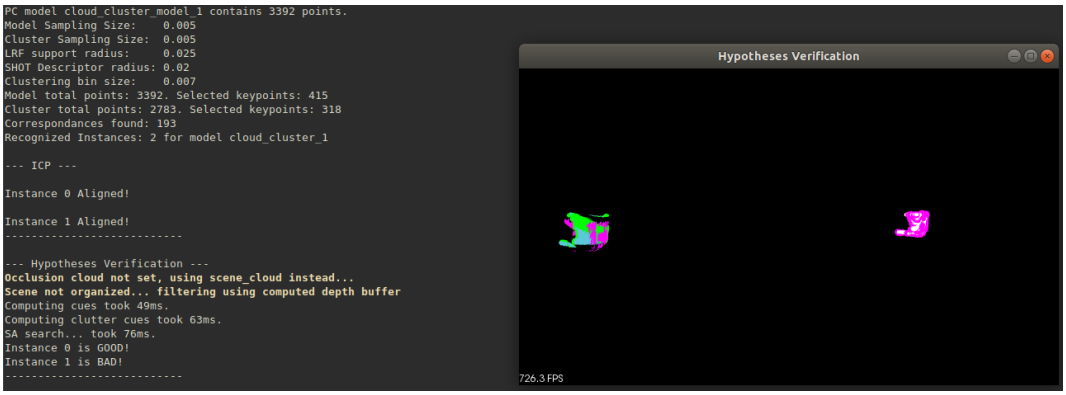

Figure 3.19: 3D Object Recognition wrt Model1 for cluster 2

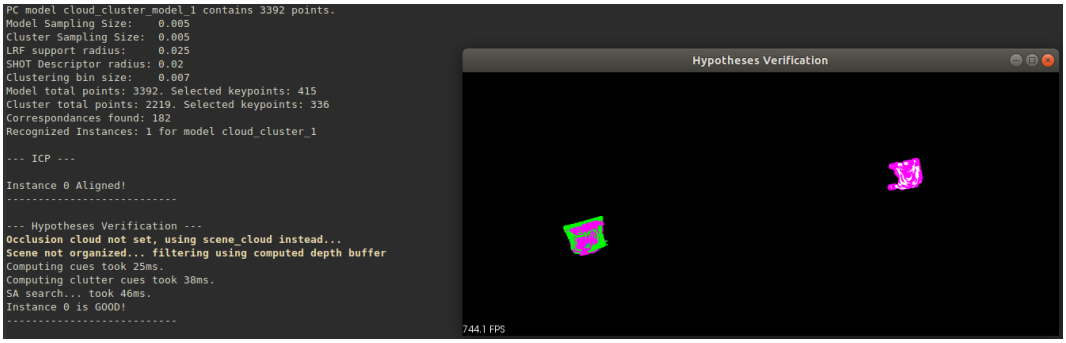

Figure 3.20: 3D Object Recognition wrt Model1 for cluster 3

Chapter 4

Planning of Robot Movements

The *MoveGroupInterface* class is a simple user interface that provides easy to use functionality for most operations that a user may want to carry out, specifically setting joint or pose goals, creating motion plans, moving robot, adding objects into the environment and attaching/detaching objects from the robot. This interface communicates over **ROS** topics, services and actions to the MoveGroup node [26].

The process of managing objects within the planning scene is of paramount importance for the purpose of this work and the process of picking up a specific object is performed passing through a particular message, the *moveit_msgs::Grasp* message. It allows the definition of the various poses and postures involved in a grasping operation and the complete documentation about this particular message can be seen at [27]. In this section only the most relevant sections of this message will be analyzed:

- trajectory_msgs/JointTrajectory pre_grasp_posture: Defines the trajectory position of the joints in the end effector group before going in for the grasp;
- trajectory_msgs/JointTrajectory grasp_posture: Defines the trajectory position of the joints in the end effector group for grasping the object;
- geometry_msgs/PoseStamped grasp_pose: Defines the pose of the end effector to attempt the grasping;
- moveit_msgs/GripperTranslation pre_grasp_approach: Defines the direction from which to approach the object and the distance to travel along a specified direction axis;

- moveit_msgs/GripperTranslation post_grasp_retreat: Defines the direction in which to move, once the object is grasped and the distance to travel along a specified direction axis.

For the place pipeline the message is structured basically in the same way of the grasp step, defining the pre, post and place configurations for placing the object in a certain point within space with a certain orientation. It can be seen at [28].

4.1 Pick Place Pipeline

To clarify the usage of the just explained messages, let's go through a simple simulated pick-place demo. It is composed by the following steps:

- Initialize the ROS node;
- Generate the planning scene;
- Pick;
- Place;
- Wait for shutdown.

4.1.1 Generate the Planning Scene

In this step the planning scene is set up for the simulation and for this purpose, some specific objects are generated with respect to the same reference (*edo_base_link*):

- A support surface on which the grasping object will be spawned. It characteristics are:
  - Pose: [x y z] = [0.6 0 0.2];
  - Orientation: [w r p y] = [1 0 0 0];
  - Dimensions: [x_{depth} x_{width} z_{height}] = [0.2 0.4 0.4].
- A support surface on which the object will be placed. It characteristics are:
  - Pose: [x y z] = [0 0.6 0.2];
  - Orientation: [w r p y] = [1 0 0 0];
  - Dimensions: [x_{depth} x_{width} z_{height}] = [0.4 0.2 0.4].
- The object to manage. It characteristics are:

- Pose: [x y z] = [0.6 0 0.5];
- Orientation: [w r p y] = [1 0 0 0];
- Dimensions: [x_{depth} x_{width} z_{height}] = [0.02 0.02 0.2].

All the described objects are added as *BOX* primitives and the coordinates in terms of [x y z] represent the position of the center of gravity (CoG) of the item with respect to the *edo_base_link* reference frame. This aspect imposes some constraints on the generation of the grasping object. As can be seen in Figure 4.1, the two support surfaces are located at the same level on the z axis, equal to $z_{CoGTable\#} + \frac{z_{heightTable\#}}{2} = 0.4 = z_{surf}$.

So according to the $z_{heightObj}$ of the grasping object, the following constraint must be respected:

$$z_{CoGObj} = z_{surf} + \frac{z_{heightObj}}{2}$$

that in this case leads to the result already described in the object CoG coordinates for the z parameter.

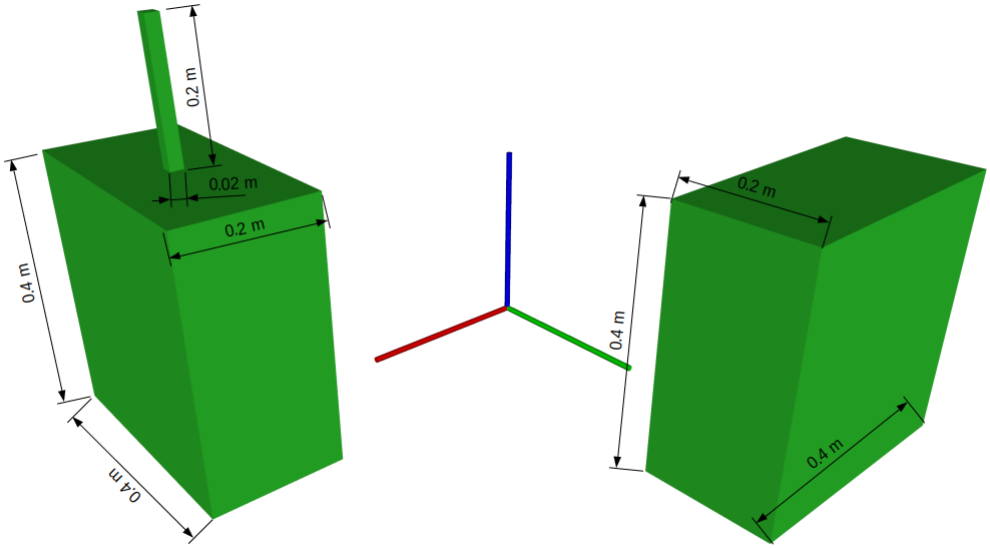

Figure 4.1: Pick Place planning scene configuration

4.1.2 Pick

Once the planning scene has been correctly configured, the pick pipeline is performed, according to the following steps:

- Generation of a vector of possible grasps (useful if used together with a grasp generator). In this case this vector is limited to a single grasp defined manually;
- Define the RF with respect to which the grasp is performed (*edo_base_link*);
- Define the gripper orientation and pose for grasping (basically in this step the grasp pose is defined, shown in Figure 4.2).

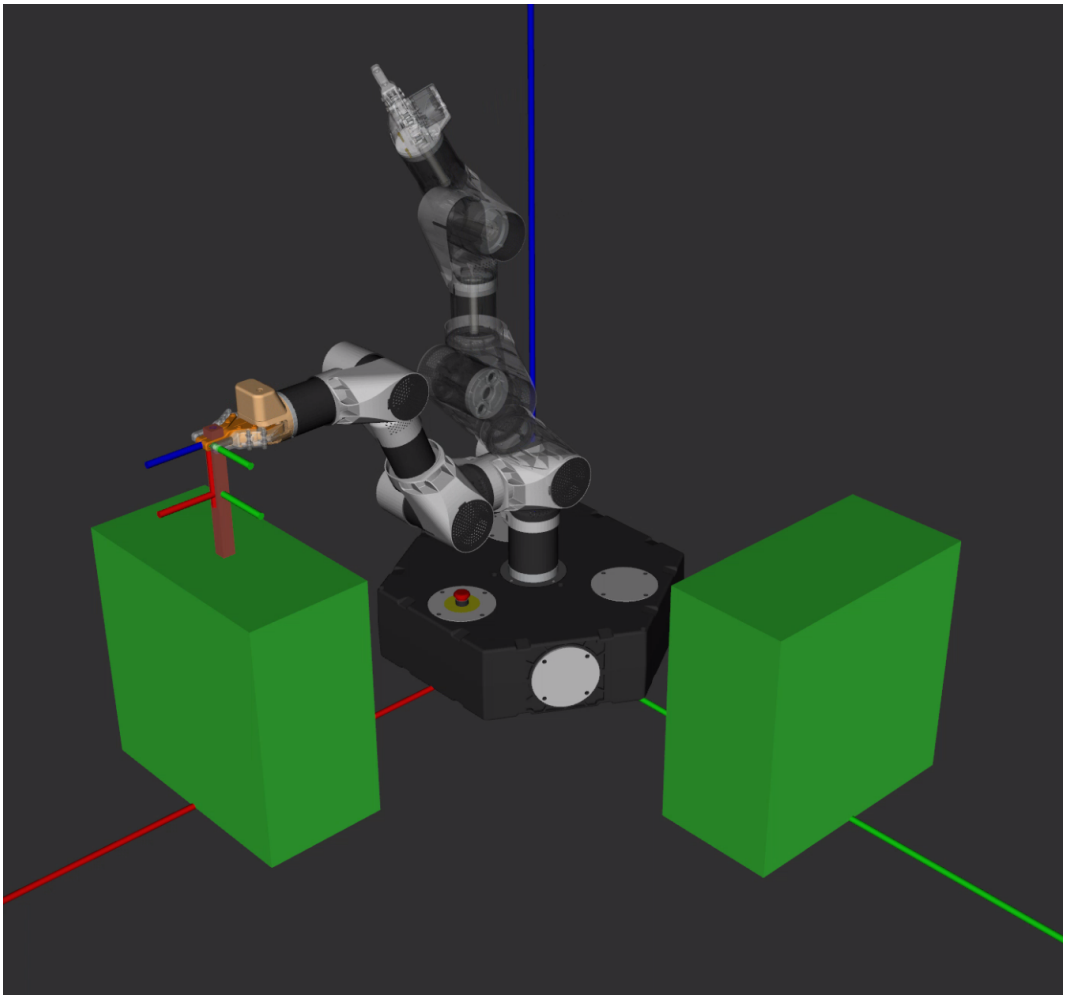

Figure 4.2: Grasp

The target object pose is generated taking into account the gripper physical limits in terms of maximum width between the two fingers and reachable positions in space of the cobot. In particular all the area represented by the support surface has been set to be completely reachable by the cobot. Moreover the gripper has been configured to have a grasp pose in the central-top portion of the target object (with respect to the z axis of the edo_base_link RF), including some extra padding with respect to the x axis of the same RF. In this way it is guaranteed that the gripper reaches a position with respect to edo_base_link RF that ensure to have the whole object within the grasping area of the fingers (i.e. to do so the TCP must reach on the x coordinate a value equal to $x = \frac{x_{depthObj}}{2} + x_{CoGObj}$).

For the orientation, the gripper has been aligned to the y axis of edo_base_link, applying a classic RPY rotation. This rotation basically represents the orientation of the edo_gripper_link_ee RF as a new rotated RF with respect to edo_base_link obtaining so that, to achieve the desired orientation a simple rotation of 90 degrees with respect to edo_base_link y axis is sufficient, as can be seen in Figures 4.3 and 4.4, taking as complete reference scenario Figure 4.5.

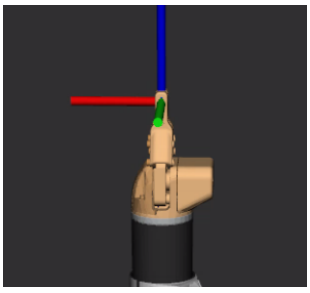

Figure 4.3: Gripper pose at rest

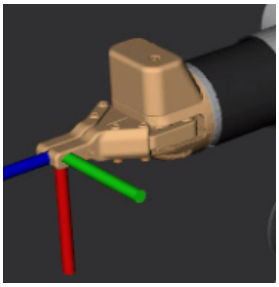

Figure 4.4: Gripper pose at grasp

It is important to notice that the gripper orientation is defined in terms of Euler angles but it needs a quaternion representation, so once defined, these angle representation is converted using the *setRPY* method of the *tf2::Quaternion* class [29];

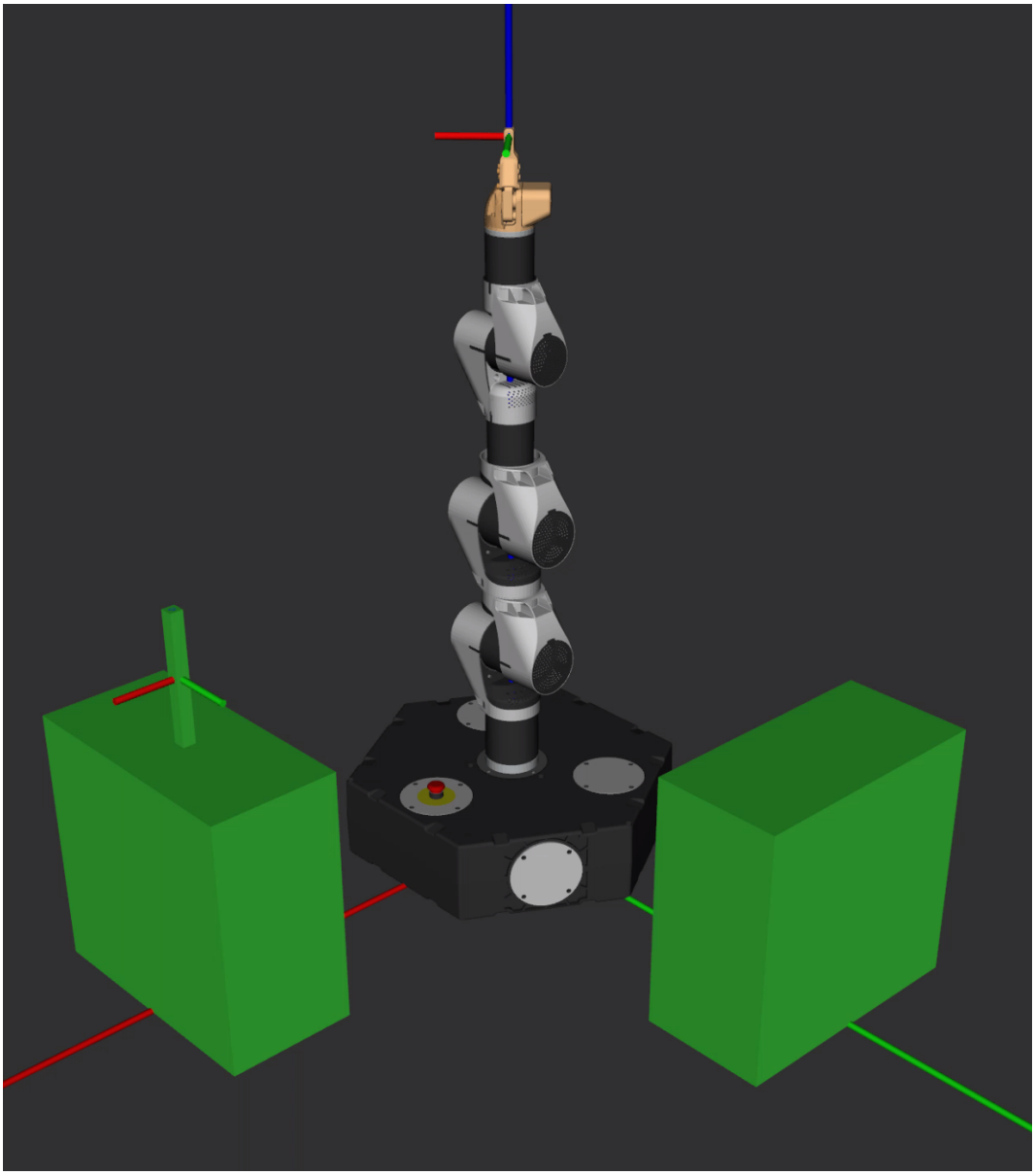

Figure 4.5: Pick Place planning scene with e.DO at rest

- Definition of the pre-grasp approach setting the grasp RF, the direction with respect to which the target object must be approached and the desired and minimum distances of the pre-grasp approach from the target object (in meters), as can be seen in Figure 4.6;

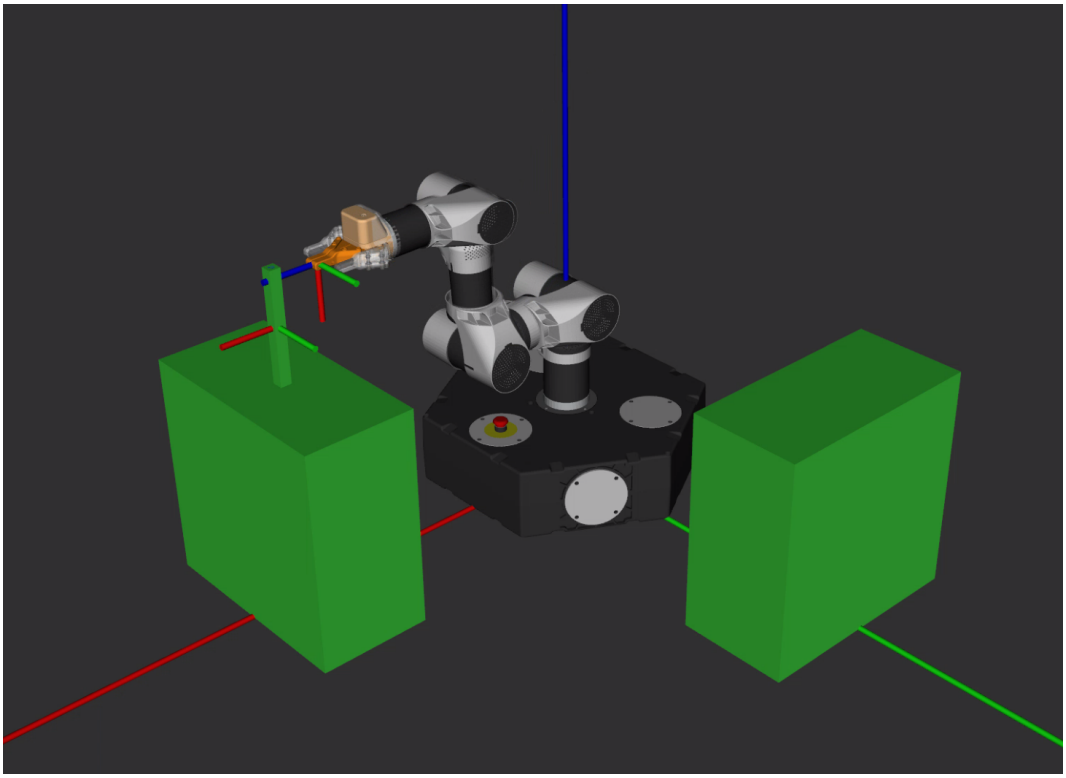

Figure 4.6: Pre-Grasp approach

- Definition of the post-grasp retreat setting the grasp RF, the direction with respect to which move the picked object and the desired and minimum distances to reach with respect to the original target object pose, as can be seen in Figure 4.7;

- Once all the pre, post and grasp configurations have been set, it is necessary to set the eef status at each different step, basically opening the gripper during the pre-grasp approach and closing it during the grasp step. Once the object has been picked it is important to keep the gripper closed, setting the new gripper status only when the target pose have been reached (i.e. during place step). Notice that both the pick and place methods need to know which is the object that must be picked and on which support surface is available or it must be brought respectively;

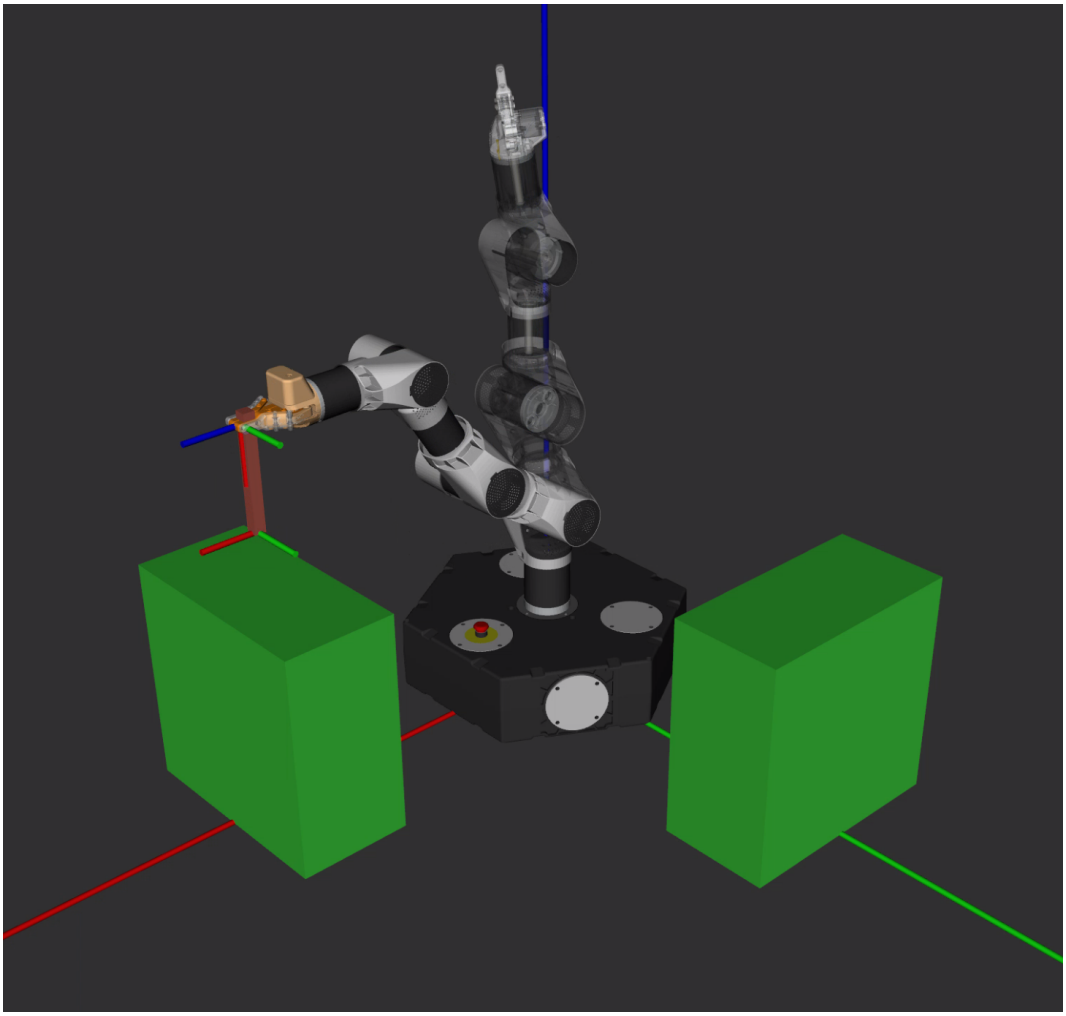

Figure 4.7: Post-Grasp retreat

4.1.3 Place

Basically this pipeline is similar to the pick pipeline, with proper changes. In the following the main conceptual changes are summarized in brief with the relative results:

- It uses another type of message called *moveit_msgs::PlaceLocation* [28], that is composed in the same way of the grasp message;
- The pre-place approach direction vector has been set toward the negative direction of the z axis of edo_base_link RF, to move the object vertically

when the place pose has been reached. In Figures 4.8 and 4.9 are reported the configurations for pre-place and place;

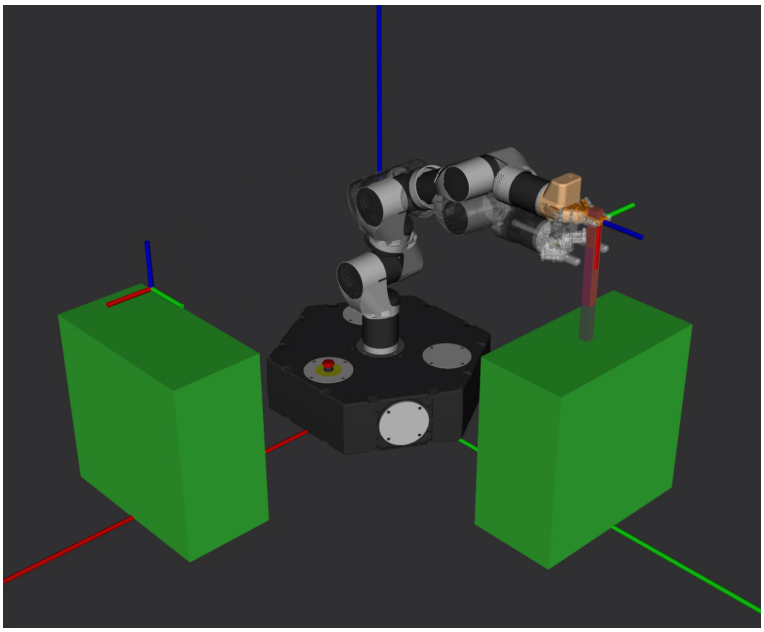

Figure 4.8: Pre-place approach

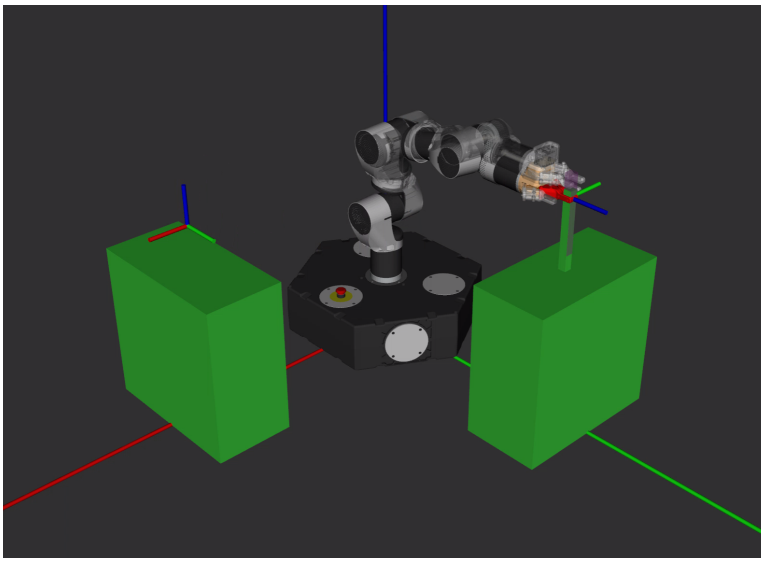

Figure 4.9: Place

- The post-place retreat direction vector has been set toward the negative direction of the y axis of edo_base_link RF to move the cobot backward, avoiding any possible collision if a new motion request incomes. In Figure 4.10 it is possible to see the post-place retreat configuration for the cobot.

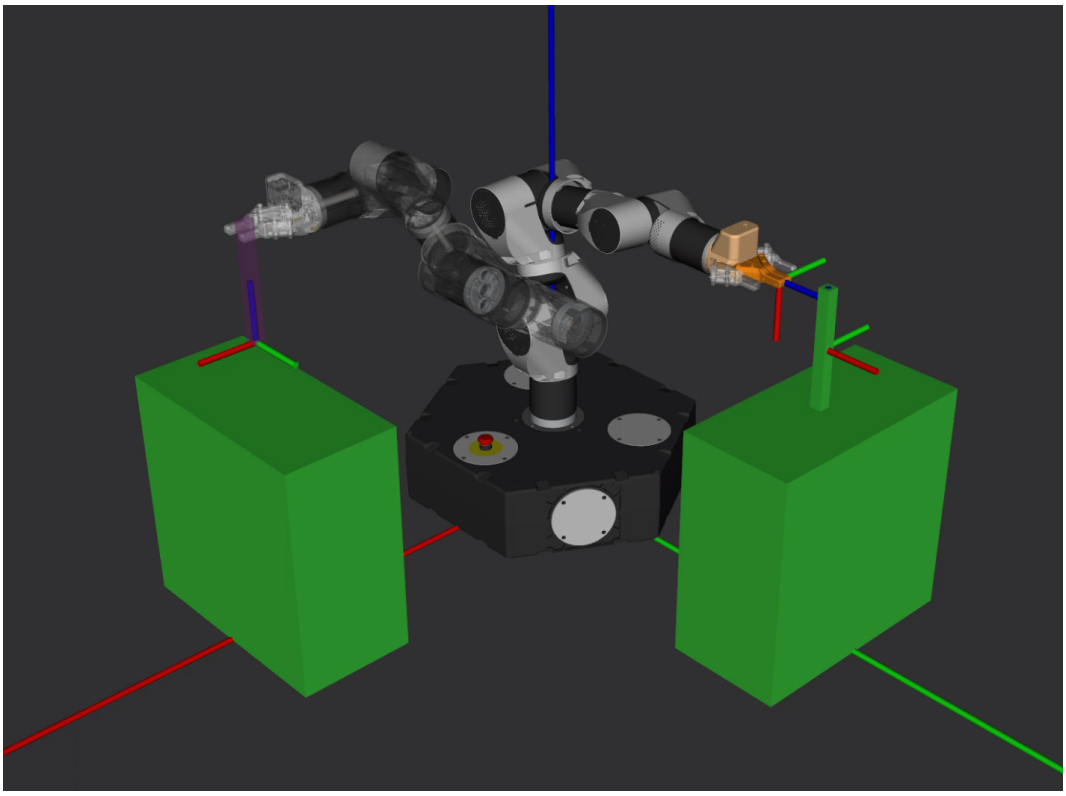

Figure 4.10: Post-place retreat

4.2 Grasp Pipeline

As previously said, for the grasp pipeline is really useful to use a grasp generator, in order to obtain different possible grasps for the same object, avoiding to hard code this step. This aspect is of paramount importance to have no limitations relative to IK or collision objects near to the target one. For this purpose the generator available in MoveIt has been rearranged to fit the necessities of the e.Do cobot. It is a generator for objects such as blocks or cylinders and provides functionality for filtering grasps based on reachabiliy and Cartesian planning of approach, lift and retreat motions.

Its algorithm is based on simple cuboid shapes and does not consider friction

cones or other grasp dynamics. Below some results about grasp computations obtained for simple objects (i.e. cuboid and cylinder) will be reported. To make more easy to understand how the grasp generator works, only the example of a simple cuboid will be handled, considering that for a cylindrical object the steps performed are exactly the same.

MoveIt grasps is based on three main components:

- **Grasp Generator**: Uses the eef kinematics and the object shape for sampling grasp poses and optimizing them, using geometric scoring functions;
- **Grasp Filter**: Validates the feasibility of grasp candidates by searching for IK solutions to verify their reachability, as shown in Figure 4.11;

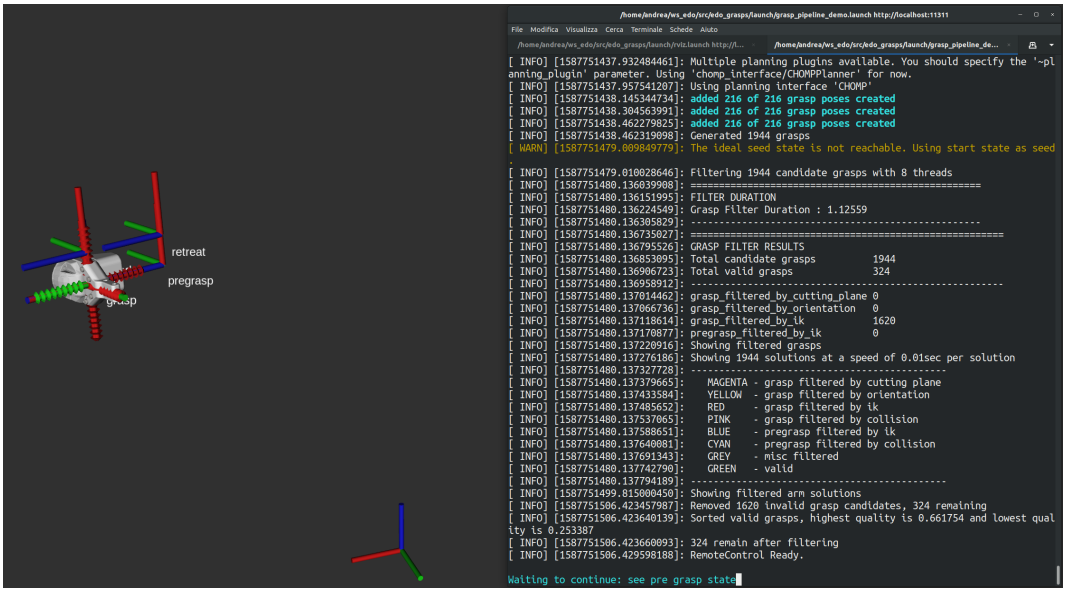

Figure 4.11: Results for grasps generation and filtering

- **Grasp planner**: Compute Cartesian approach, lift and retreat trajectories that compose a complete grasp motion.

To work correctly, these three components need to be applied in sequence. Additionally, the grasp generator uses a **Grasp Scorer**. It is a component that supports a number of heuristics for judging which grasp are favorable, given known information about the problem/application. In Figures 4.12, 4.13, 4.14 and 4.15 all these four configurations are shown for the case of a simple cuboid (at mid-air) with no obstacles in the planning scene.

The next section, will describe some configuration notes, chosen for the specific robot used during this work.

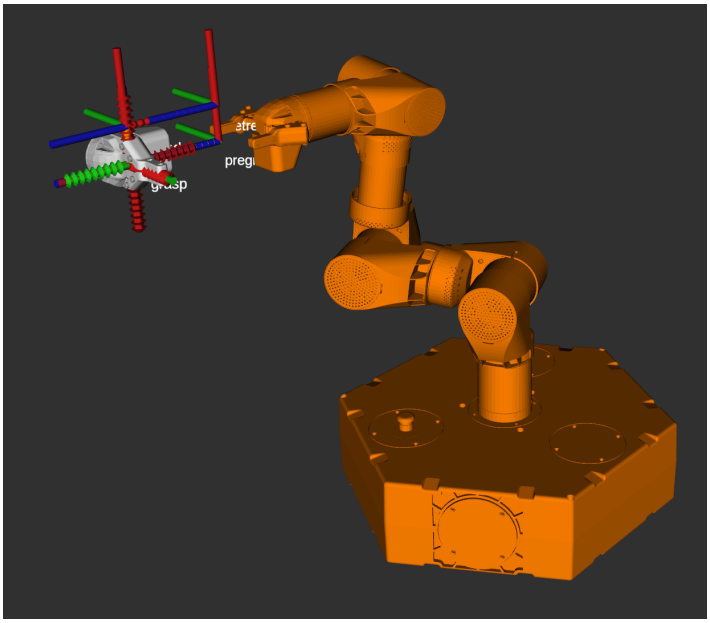

Figure 4.12: Pre-grasp approach

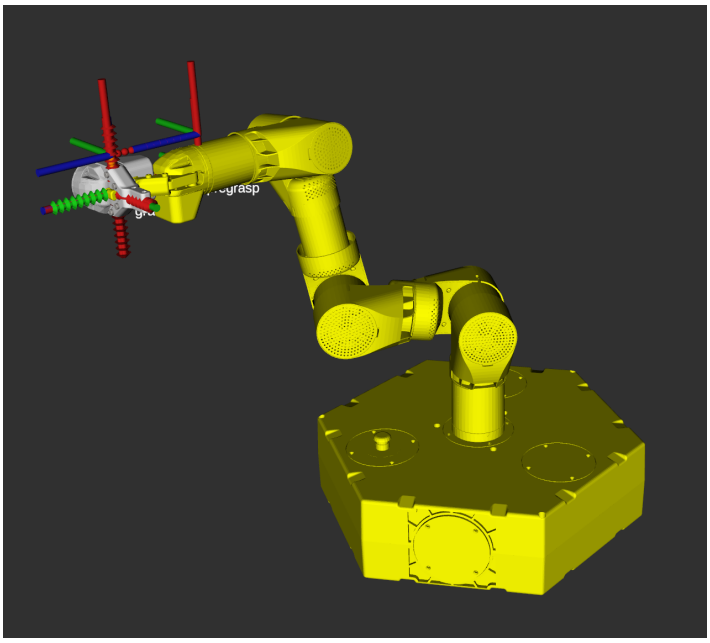

Figure 4.13: Grasp

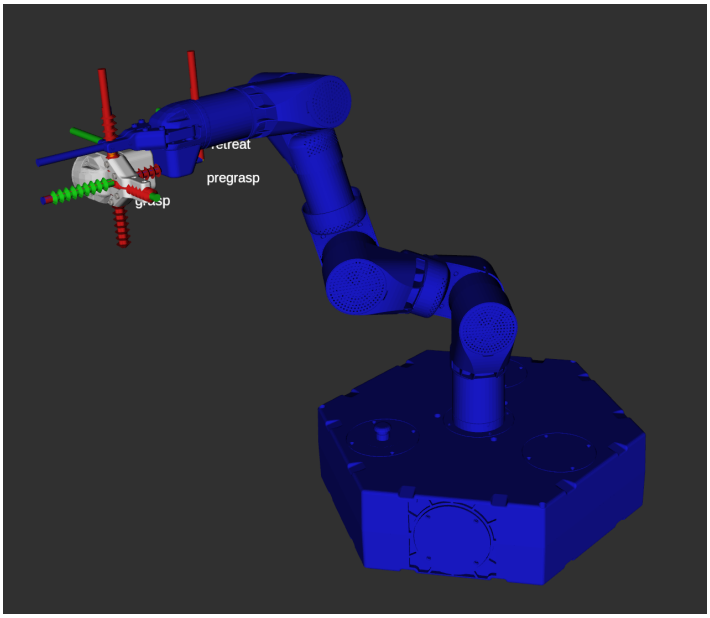

Figure 4.14: Post-grasp lift

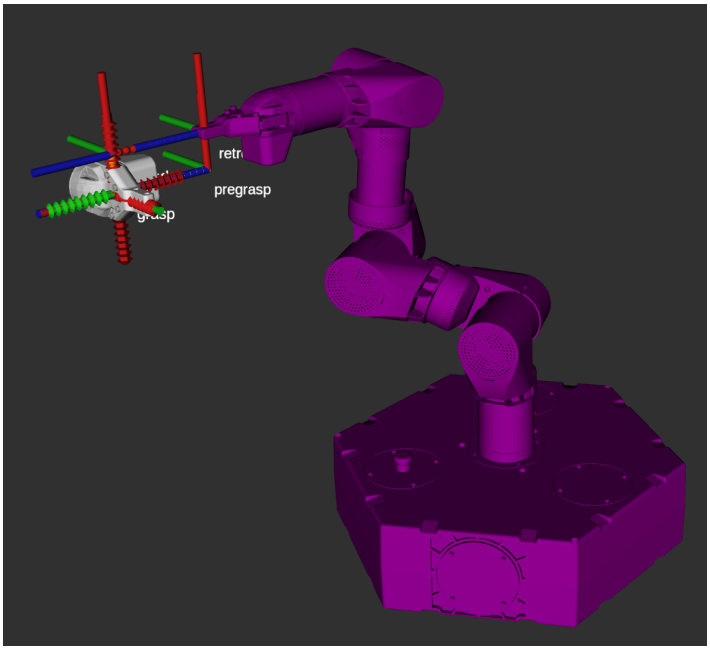

Figure 4.15: Post-grasp retreat

4.2.1 Robot-Agnostic Configuration

The robot configuration can be loaded through the *.yaml* configuration files as rosparams with the grasping application/ROS node, as shown in the following code snippet:

```
<node name ="edo_grasps_demo" pkg="edo_grasps"
type="edo_grasps_grasp_pipeline_demo">
  <param name="ee_group_name" value="edo_gripper" />
  <param name="planning_group_name" value="edo" />
  <rosparam command="load" file=$(find edo_grasps)/config_robot
                                    /edo_grasp_data.yaml" />
  <rosparam command="load" file=$(find edo_grasps)/config
                                    /edo_grasp_config.yaml" />
</node>
```

where:

- *ee_group_name* specifies the robot eef group;
- *planning_group_name* specifies the robot planning group.

4.2.2 Additional Configuration

- **tcp_to_eef_mount_transform**: Represents the transform from the tool center point used for grasp poses to the mount link of the end effector. This parameter is provided to allow different *URDF* end effectors to all work together without recompiling the code.

  In MoveIt, the actuated end-effector fingers should always have a parent link, typically the wrist or palm link. The wrist link should have its palm with a z axis pointing towards the object to grasp (i.e. where the pointer finger is pointing). This is the convention laid out in 1955 by *John Craig* in *Robotics*, but considering that a lot of *URDF* models do not follow it, this transform allow to fix the problem. Additionally the x axis should points up along the grasped object, meanwhile the y axis should point towards one of the fingers;

- **Switch from Bin to Shelf Picking**: In grasp generator two methods can be used to select an ideal grasp orientation for picking:

  – setIdealGraspPoseRPY ();

  – setIdealGraspPose ();

These two methods are used to score grasp candidates favoring grasps that are closer to the desired orientation. This is useful in applications such as bin and shelf picking where the pick step for an object, needs to be done from a bin with a grasp that is vertically aligned and from a shelf with a grasp that is horizontally aligned. In the case of this e.DO cobot the favourite orientation for the gripper has been set to be [r p y] = [0 90° 0] with respect to world RF (that coincides with edo_base_link RF). In this way all the remaining grasps from filtering step, are sorted to put as favourite solutions all the grasps that ensure a gripper orientation as the one shown in Figure 4.4.

- The gripper of the e.DO cobot moves the fingers according to an angular opening of the joint, but the MoveIt grasp pipeline is able only to manage gripper with linear finger opening expressed in meters, meanwhile the e.DO gripper requires a fingers opening in radiant. To solve this problem a new function has been added to the MoveIt grasp class to convert the finger opening from meters to radiant, as shown in Figure 4.16.

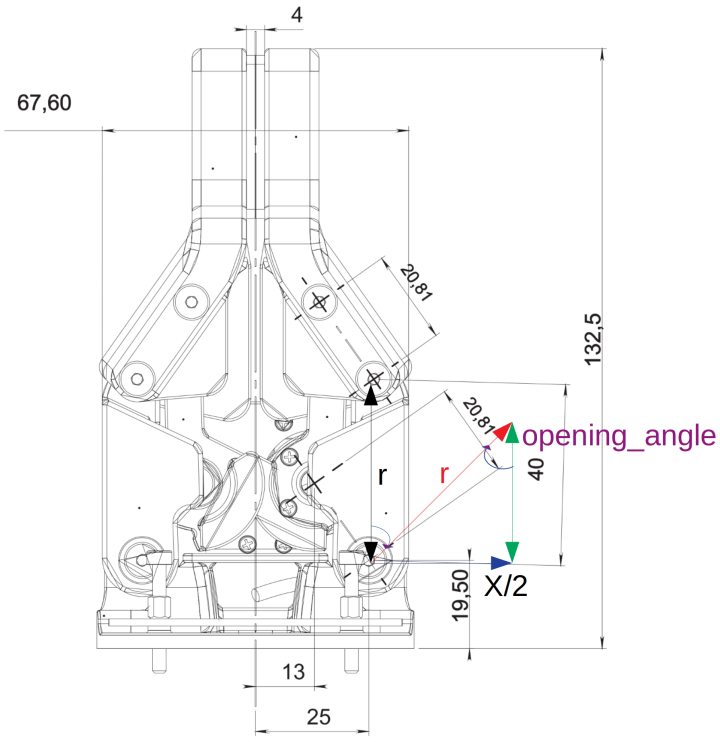

Figure 4.16: Conversion from linear to angular finger opening

The result can be easily obtained applying the trigonometric theorem:

$$\frac{X}{2} = r \sin(openingAngle) \rightarrow openingAngle = -\arcsin\frac{X}{2r}$$

The negative sign is due to the fact that the gripper opens with a clockwise rotation, that is negative for convention and the value of X represents the maximum width between fingers. The class has been modified just adding two variables (modifying the two files *two_finger_grasp_data.cpp* and *two_finger_grasp_data.h*):

- bool variable to chose the gripper type (false means linear opening, angular otherwise);
- double variable that contains the result of the conversion function.

In this way is possible to select the gripper type and to provide the function result just adding these two lines in the code:

```
//Select the gripper behaviour
grasp_data_->ee_type = false;
//Set the values of width and distance from TCP to convert
//linear (m) distance in angular opening (rad)
grasp_data_->distance_btw_fingers_rad = widthToAngle(0.082,
                                                     0.04);
```

Notice that as a default setting, if noting is specified in code, the class works with the original linear opening settings provided by the original MoveIt class.

Chapter 5

Demonstrator Design and Results

In this section, the design implementation to realize the final demonstrator will be explained. The main structure can be summarized with the following steps:

- Setup planning scene;
- Select the target objects method: if the perception node is up, use it to generate the target objects, otherwise generate it randomly on the support surface, according to some specific constraints;
- For all the available target objects in the planning scene perform:
  - Grasps computation;
  - Pick pipeline;
  - Wait for input command: According to the input command, the picked object will be placed to a specific position in space, with a certain orientation;
  - Remove the target object already managed;
- Move the robot to its idle state;
- Wait input command to start a new acquisition or to kill or the nodes relative to this demo.

5.1 Setup Planning Scene

This section is performed only once and its aim is to prepare the planning scene, adding the support surfaces necessary to realize the demo. Here the parameters of the support surface on which the object is generated (i.e. CoG, x_{depth}, y_{width} and z_{height}) are stored for the grasps computation. This is necessary for the grasps computation, to filter out all the generated grasps that would cause a collision between robot and support surface.

The planning scene components are:

- *Table*: support surface on which the target objects are generated. Its characteristics are:
  - *Position*: [x y z] = [0.6 0 0.2];
  - *Orientation*: [w x y z] = [1 0 0 0];
  - *Dimensions*: [x_{depth} y_{width} z_{height}] = [0.2 0.4 0.4];
- *faultyBox*: support surface on which the target objects is placed if it is recognized as faulty. Its characteristics are:
  - *Position*: [x y z] = [0 -0.6 0.1];
  - *Orientation*: [w x y z] = [1 0 0 0];
  - *Dimensions*: [x_{depth} y_{width} z_{height}] = [0.4 0.2 0.2];
- *correctBox*: support surface on which the target objects is placed if it is recognized as OK. Its characteristics are:
  - *Position*: [x y z] = [0 0.6 0.1];
  - *Orientation*: [w x y z] = [1 0 0 0];
  - *Dimensions*: [x_{depth} y_{width} z_{height}] = [0.4 0.2 0.2];

The final setup can be seen in Figure 5.1.

5.2 Object Generation

Once the planning scene is ready, it is possible to select the target objects generation through a bool variable directly set in the shell, during the *.launch* file execution. This variable starts the perception node and allows the generation of the target objects through it if true, otherwise the perception node is not launched and the target object generation task is left to the random generator.

Notice that it is possible to select the target object generation method only when the node is started up, it is not possible to modify it during the execution.

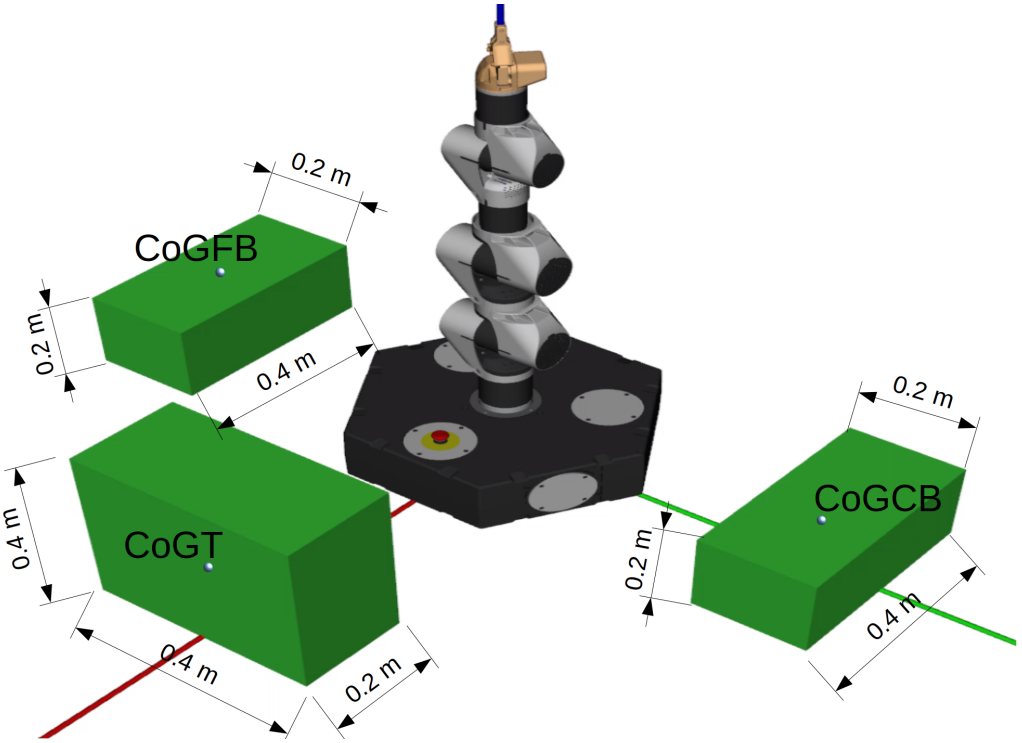

Figure 5.1: Demo planning scene setup

5.2.1 Random Generator

If the bool variable is not set, as default it is assumed as false and the target object is generated through a random generator, respecting some specific constraints. Since the robot is able to reach each position on the upper surface of the spawn support surface, the only constraints that must be imposed are the ones that guarantees to obtain the target object entirely within the support surface and in contact with it. This means to impose:

- r $<$ 0.04 meters. This constraint has been set according to the maximum width of the gripper;
- h: There is no particular constraint on the object height but, considering that the target objects generated by the perception node have an height value between 0.05 and 0.07 meters, to make the simulation as realistic as possible, this value has been set as upper and lower limits;

- CoG_{Obj}:
  - x ϵ (0.5 + r; 0.7 - r);
  - y ϵ (-0.2 + r; 0.2 - r);
  - z = 0.4 + $\frac{h}{2}$.

The final result can be seen in Figure 5.2.

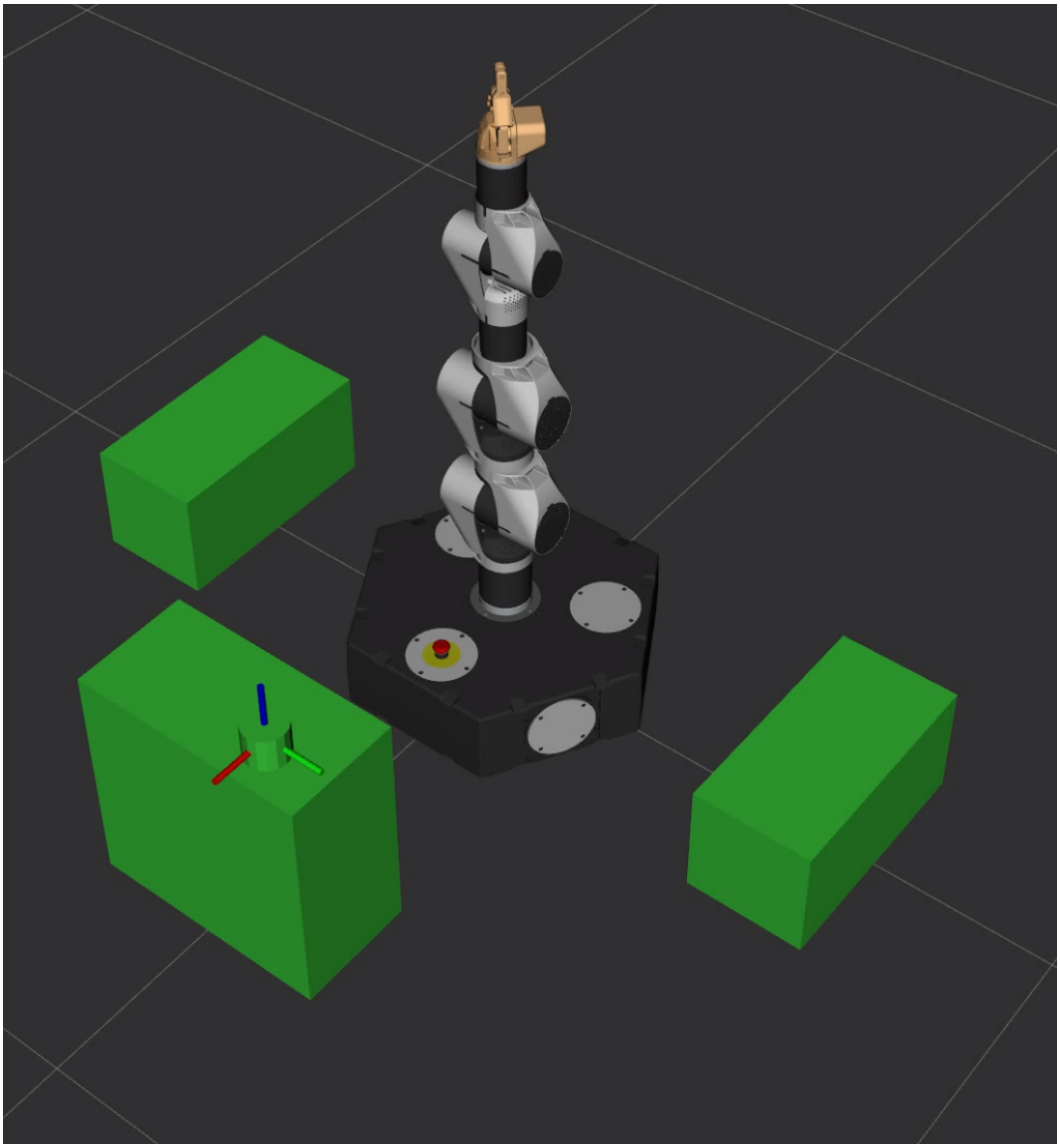

Figure 5.2: Demo planning scene setup with random generator

5.2.2 Perception Generator

If the bool variable **useCam** is set to true, the perception node is started up and the target objects are generated through the perception pipeline. For this case it is necessary to add some delay in order to guarantee the generation of all the target objects before starting the execution. In this way all the objects are considered avoiding to neglect one or more of them.

It is important to notice that the perception pipeline is always up during the demo execution, but it is executed only when the following particular conditions are respected:

- There is no object in the planning scene whose name starts with "*cylinder*";
- There is no attached object to the robot in the planning scene.

Through this solution, the perception node is executed only when all the target objects have been processed and removed. This design choice was made to improve efficiency and reduce the computational footprint, since it is useless to search for other target objects, if the planning scene is already populated. The final result can be seen in Figure 5.3.

Obviously, in this particular case, the camera position with respect to the *world* RF is set so that the objects sensed are generated and added on the spawn support surface, respecting the same constraints imposed for the random generation. This can be done specifying the pose of the camera in terms of coordinates [x y z R P Y].

Moreover, the octomap has been limited to avoid the introduction of useless constraints for the robot movements (i.e. it is considered as a collision object for the planning computation, so it has a paramount importance for the computation of the inverse kinematics solution).

5.3 Target Objects Management

Once at this step, the grasp and pick-place pipelines are executed, but before going on, let's point out some specific design choices taken for the random and perception generators:

- For the random generation, the number of target objects has been limited to a single one;
- For the perception generation, the number of target objects has no limits and all the objects sensed will be managed serially. Anyway it is important to respect the constraints imposed by the *Euclidean Cluster Extraction* method, which

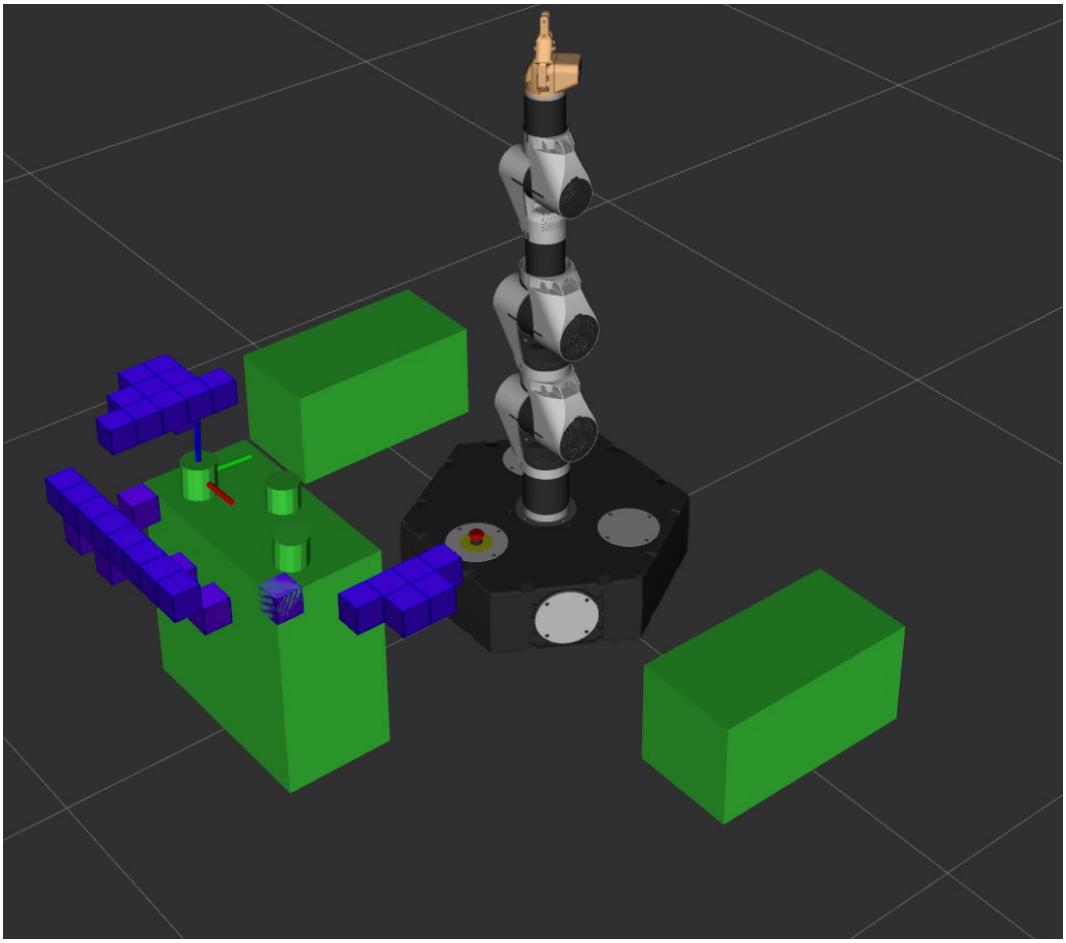

Figure 5.3: Demo planning scene setup with perception generator

impose to have the different objects at least at 2 centimeters, to avoid that different objects are interpreted as a single cluster. Moreover, an additional control sequence has been added for objects whose plan fails at the first computation.

Basically, if an object planning fails, and it has been processed only once, its characteristics are pushed back to the tail of the object vectors and all the other objects are processed. Once all the movable objects have been correctly managed and removed, a new grasps and planning trial are executed to check whether the failed planning objects are not reachable or if they were initially not reachable due to presence of collision. If at the second computation they can be managed, they are picked and placed accordingly to the default chain. On the other hand, if the planning fails again, probably these objects are not

reachable by the robot and so they are removed from the planning scene. Here the code snippet that manages this control is shown:

```
/* If the object cannot be picked, it is stored
 * at the vector tail to be processed again.
 * This can be done basically checking if the
 * iterator is greater than the maximum_iteration
 * value. */
 if (iterations == MAX_ITERATIONS && !pickOK){
   object_name.push_back(object_name[obj_cnt]);
   object_pose.push_back(object_pose[obj_cnt]);
   radius.push_back(radius[obj_cnt]);
   height.push_back(height[obj_cnt]);
   doNotRemove = true;
   for (size_t i = 0; i < obj_cnt_original; ++i) {
     if (object_name[obj_cnt] == object_name[i]) {
       object_occurrences[i] += 1;
       if (object_occurrences[i] > 1){
         // The object has been processed two times,
         // so it can be removed
         doNotRemove = false;
       }
     }
   }
 }

 ...
 ...
 ...

if (!doNotRemove) {
  removeObject(planning_scene_interface,
               object_name[obj_cnt]);
}
```

Notice that, calling the .push_back() method within the for cycle that scans all the available objects, each time an object is added to the tail the for cycle iterations are incremented by one element. This is good to dynamically manage all the original and the pushed back objects whose planning fails, but in this way there is no control on the number of times the same object has

been processed. To solve this problem, a new vector declared outside the for cycle is used. It is initially resized to the original object vector dimension and it stores the occurrences of each element. In this way, checking the number of occurrences of the physically available target objects, it is possible to decide if a new computation is necessary or if the object is not reachable and so can be removed. Moreover the number of times the planner is called has been fixed to a maximum value and also the time to compute the planning has been increased. This is due to the particular planner used for this purpose, the **OMPL** planner, that is not the most powerful planner available but for the moment is the most convenient one that works correctly with all the components of the e.Do cobot. For more information about this planner, please take a look to [30].

5.3.1 Grasps Computation

Once the planning scene has been correctly set up, and the target objects have been added, randomly or thought the perception node, the grasps generator is called taking the following inputs:

- Object information:
  - object_pose[i] (position and orientation);
  - radius[i];
  - height[i];
  - object_name[i];
- Support surface information:
  - table_pose (position and orientation);
  - x_{depth};
  - y_{width};
  - z_{height};
  - table_name.
- Vector of the resultant available grasps (moveit_msgs::Grasp).

The value returned by the grasps generator is used to check if at least a solution is available after filtering. If true, then the pick-place pipelines can be performed, meanwhile if false, the object is removed and the computation is brought back to the target objects generation. This aspect is useful to define the resolution to use for the grasps generation.

Grasps are generated on all the object's faces, edges and along the x and y axis with variable angle. Notice that also if the target object is a cylinder the grasp generator works on the bounding box of the object, with dimensions [radius radius height] allowing to generate grasps not only along the selected axis with variable angle, but also each 45 degrees with respect to the object z axis.

All the parameters relative to the grasp generator can be set within the "*edo_grasp_data.yaml*" file, highlighting that:

- **grasp_resolution** represents the translation, in meters, between one grasp candidate and the other;
- **angle_resolution** represents the angular rotation, in radiant, between one grasp candidate and the other with respect to the same axis;
- **tcp_to_eef_mount_transform** represents the distance, in meters, from the eef mount to the palm of the end effector, where the z axis points toward the object to grasp, the x axis is perpendicular to the movement of the gripper and the y axis is parallel to the movement of the gripper;

5.3.2 Pick-Place

The pick and place pipeline are performed basically as already described in the previous chapter, introducing the keyboard input for the latter and a correct planning check for both. In fact both the "*.pick()*" and "*.place()*" methods returns an error code if the planning fails. Converting this error code into a bool variable, it is possible to check if the pick is performed correctly (if the object fails for more than once, as already described, it is removed and the input command wait, together with the place pipeline, are skipped). Similarly, if the place pipeline fails, the object is removed, paying attention to the fact that once picked, the collision object becomes an attached object, so it is necessary to detach it from the robot before deleting.

Notice that, with respect to what explained for the pick place in the previous chapter, in this case both these methods takes a vector of possible **Grasp** and **PlaceLocation**. While the first is the result of the grasps generator after filtering, for the place pipeline the vector is generated considering some specific aspects of the particular application. Since the aim of the demo is just to separate the correct from the faulty objects, there are no particular needs on the object orientation, limiting the place pipeline to respect only the target position. Considering what just pointed out, and considering that the objects sensed from the perception pipeline are generated with the z axis parallel to the z axis of the *edo_base_link* RF, is it possible to assume that at the target position, the object can assume a random orientation with respect to its z axis.

For this reason, the place vector has been generated as a variable length vector, whose dimension depends on the angular resolution that the final user wants to have, with respect to the object's z axis, in degrees.

It is also important to highlight that in the final demo, the input commands will not be provided from keyboard (**K** → Correct and **F** → Faulty), but they will be represented by a KWS command (among the KWS commands available from the KWS algorithm), received serially from the SensorTile board. This temporary solution was imposed by the pandemic situation and will be fixed as soon as it will be possible to make some tests on the real board. In Figures 5.4 and 5.5 the positions assumed by the robot for the **K** and **F** configurations are shown.

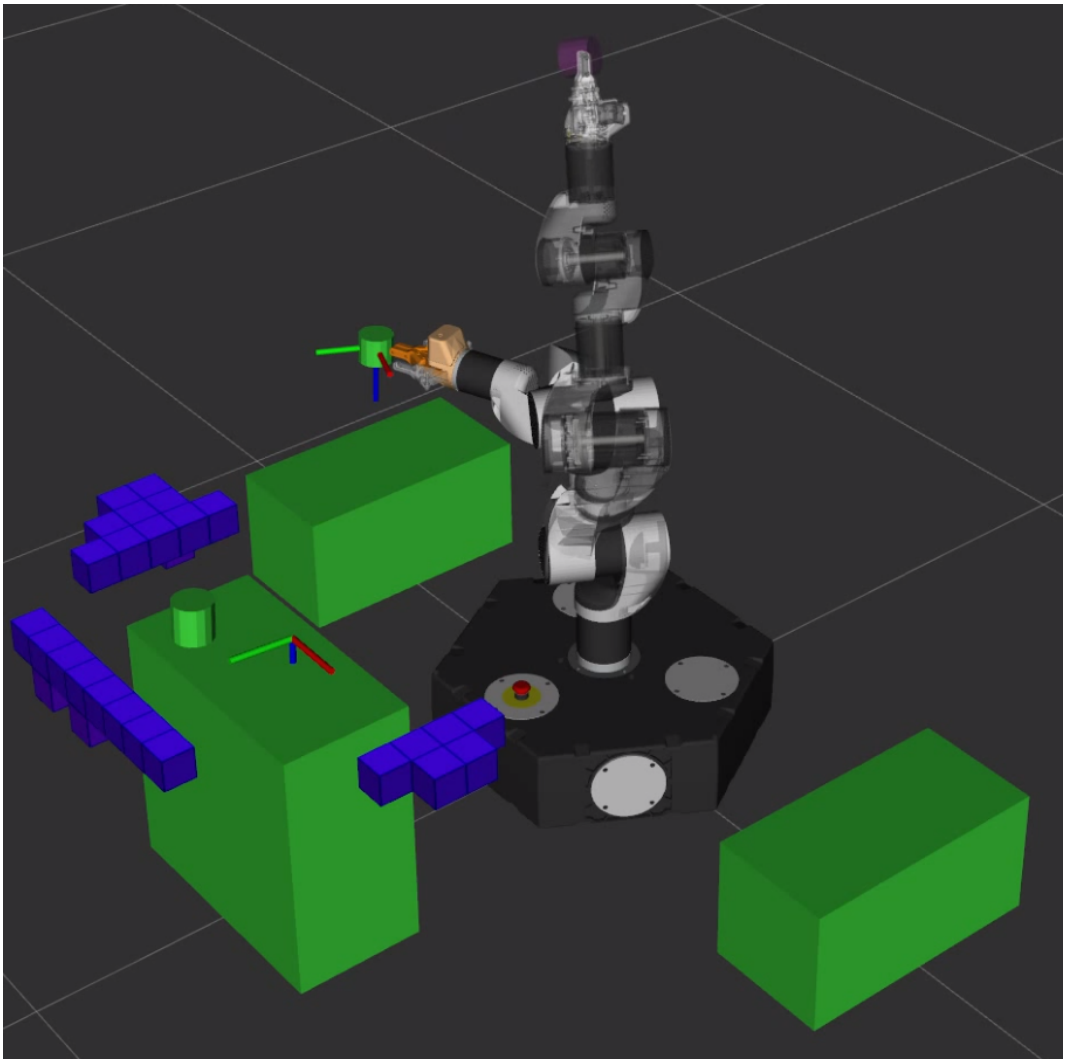

Figure 5.4: Faulty Placing

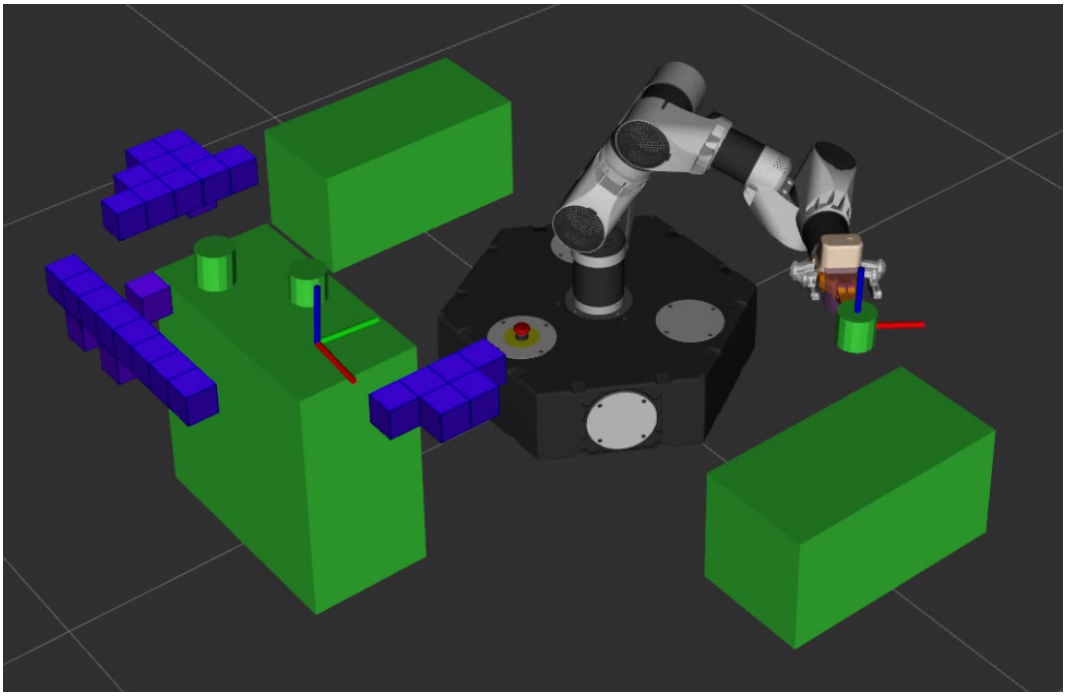

Figure 5.5: Correct Placing

5.4 Return to Idle State

Once all the objects have been managed and removed, the perception node (if started) add the new sensed objects, and the robot is moved to its rest state. To do so it is sufficient to set:

- For the "**edo**" move_group:
  - joint_1 = 0.0;
  - joint_2 = 0.0;
  - joint_3 = 0.0;
  - joint_4 = 0.0;
  - joint_5 = 0.0;
  - joint_6 = 0.0;

- For the "**edo_gripper**" mouve_group: joint_gripper_left_base = 0.0;

The values of the joint are expressed in radiant and for the gripper group only the *joint_gripper_left_base* is modified since it is the only revolute joint.

At this point th computation is stucked, waiting a new input command from keyboard, considering that:

- Pressing **Enter** a new computation is performed;
- Pressing **q** the computation is stopped and both the demo and perception nodes are killed.

5.5 How to Use It

To start the demo, according to the presence of the perception node, type on different shells:

- *roslaunch realsense2_camera rs_camera.launch filters:=pointcloud*

  Only if the perception pipeline is used and the realsense camera is plugged to a USB port;

- *roslaunch edo_scenarios rvizConfig.launch simulated:=true*

  If the real robot is connected, omit the argument *simulated:=true*, since its default value is false;

- *roslaunch edo_scenarios demo.launch useCam:=true*

  To use the random target object generation, remove the *useCam* argument, since its default value is false, and do not run the launch file of the first shell.

Chapter 6

Gazebo Simulation

Gazebo is a robot simulation tool that allows to rapidly test algorithms, design robot, perform regression testing, and train AI system using realistic scenarios.

It offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments.

Basically it is a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.

The main features of this simulator are:

- **Dynamic Simulation**: Access multiple high-performance physics engine including ODE, Bullet, Simbody, and DART;
- **Advanced 3D Graphics**: Utilizing OGRE, Gazebo provides realistic rendering of environments, including high-quality lightning shadows, and textures;
- **Sensors and Noise**: Generate sensor data, optionally with noise, from laser range finders, 2D/3D cameras, kinect style sensors, contact sensors, force-torque, and more;
- **Plugins**: Develop custom plugins for robot, sensor, and environmental control. Plugins provide direct access to Gazebo's API;
- **Robot Models**: Many robots are provided including PR2, Pioneer2 DX, iRobot Create, and TurtleBot. It is also possible to build a custom robot using SDF description;
- **TCP/IP Transport**: Run simulation on remote servers, and interface to Gazebo through socket-based message passing using Google Protobufs;
- **Cloud Simulation**: Use CloudSim to run Gazebo on Amazon AWS and GzWeb to interact with the simulation through a browser;

- **Command Line Tools**: Extensive command line tools facilitate simulation introspection and control.

For more information about Gazebo, have a look to [31].

6.1 Preliminary Settings

Before showing the simulation results, it is important to highlight some configuration aspect to make the simulation works correctly:

- **Camera plugin**: Since the Intel RealSense D435i cam is not available, the camera model and plugin has been added directly as a *xacro:macro* part of the robot model developed for the simulation. This sensor has been moved from the *edo_base_link* RF according to the following transform [1.1 0 0.55 0 0 3.14]. Notice that in this simulation, the sensor does not include any kind of noise;
- **World definition**: The Gazebo simulation is opened with a custom *.world* file that contains all the information about the item pose, in terms of visual and collision objects. For the purpose of this demo, in the custom world, has been added the models of all the grasping objects, the support surface and the dropboxes;
- **Gripper Plugin**: Since the gripper is actuated through a position controller, during grasping this is not able to sense the presence of an object between fingers, producing a not realistic performance. To solve the problem, some precautions must be taken into account.

  Firstly the *Gazebo Grasp Fix Plugin* [32], has been added within the robot description model. It basically fixes an object which is grasped to the robot hand to avoid problems with physics engines and to help the object staying in the robot hand without slipping out.

  Moreover the actuation through a position controller, cause a non-realistic behaviour for the grasp, because it tries to reach the target fingers position without taking into account the presence of the grasped object. To solve this problem, the gripper joint that actuate the fingers, takes as target pose the equivalent opening of the fingers (in radiant) that correspond to the grasping object width.

  In this way the gripper is closed around the grasping object and the plugin fixes it to the gripper creating a realistic grasp behaviour for the simulation.

Finally it was necessary to correctly calibrate the grasp configuration file to ensure a correct behaviour during grasping.

6.2 Simulation Results

Figures 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, show the obtained results:

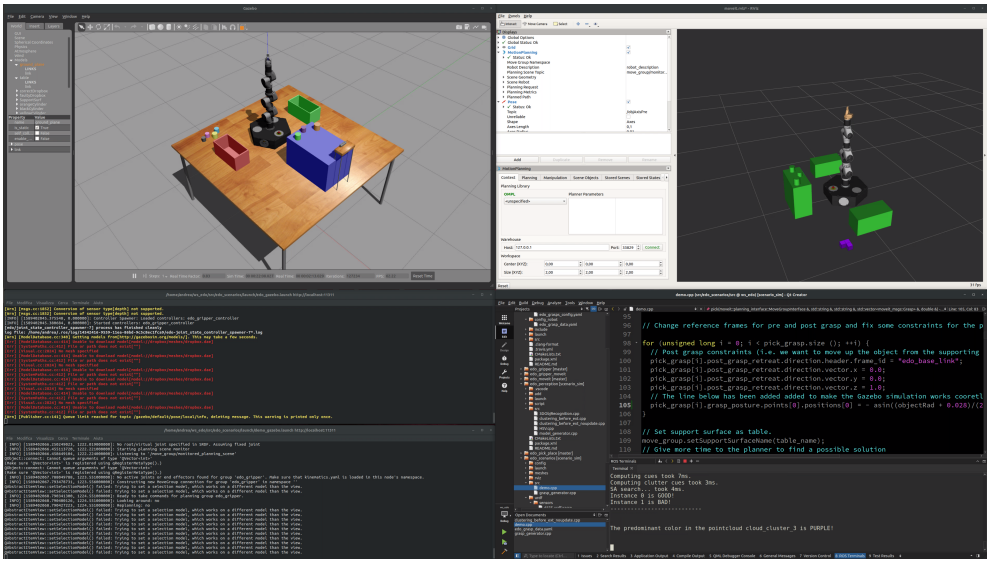

Figure 6.1: Start simulation: cylinders have been sensed and added to the planning scene.

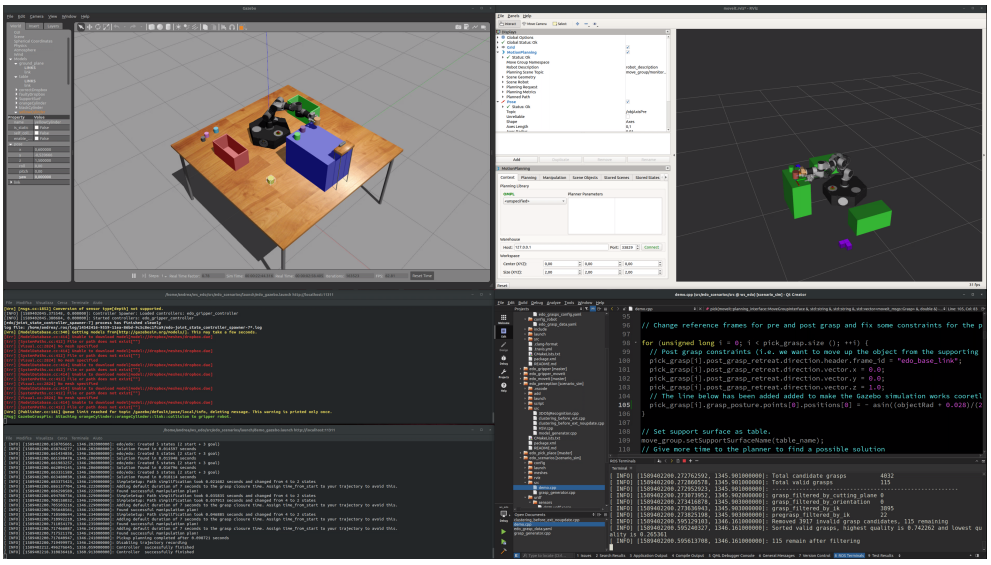

Figure 6.2: First object Pick.

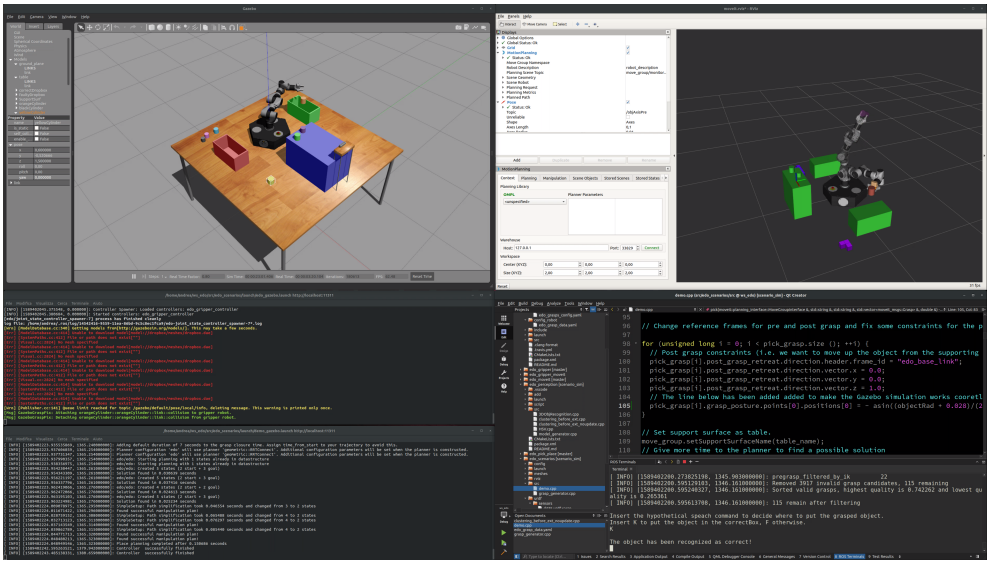

Figure 6.3: First object Place.

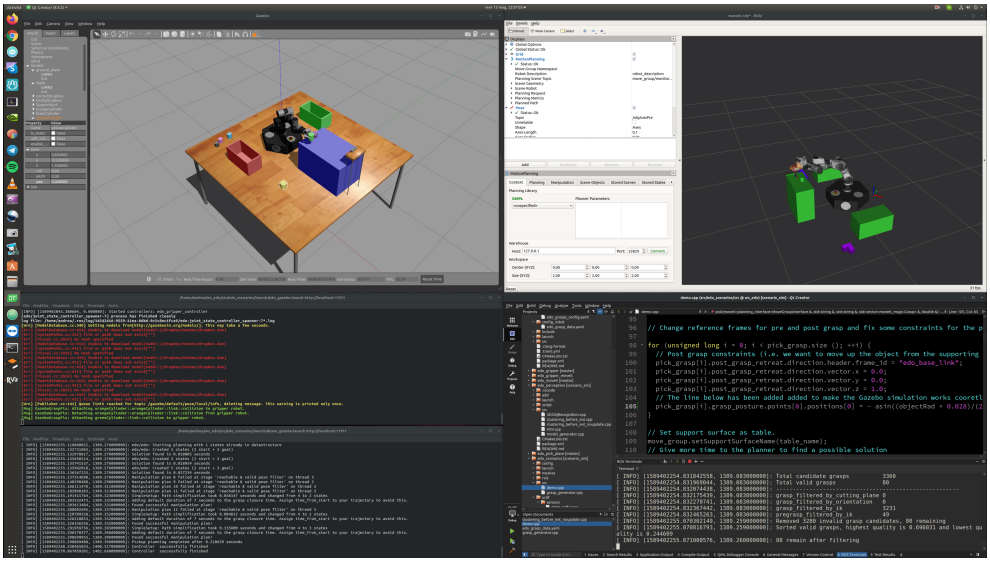

Figure 6.4: Second object Pick.

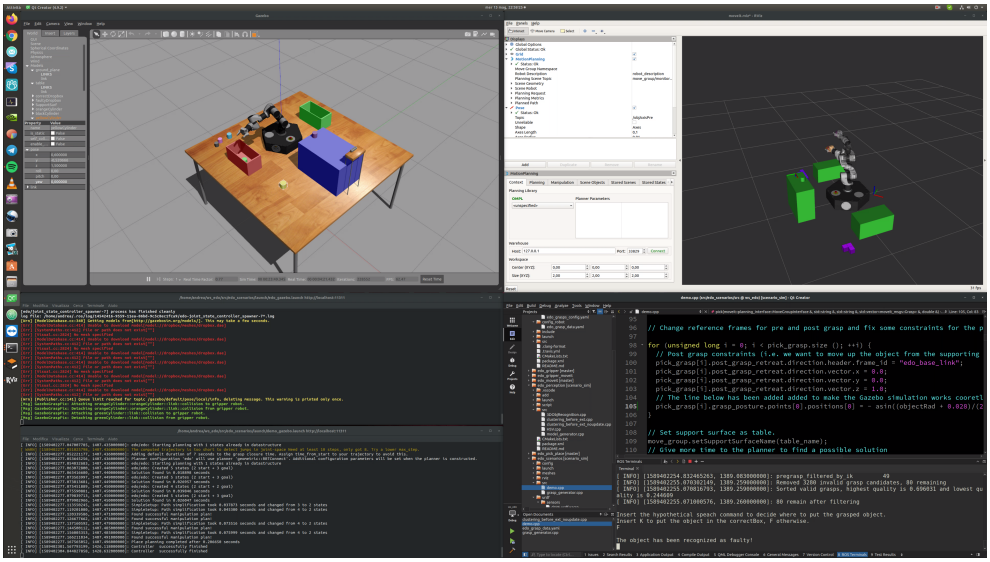

Figure 6.5: Second object Place.

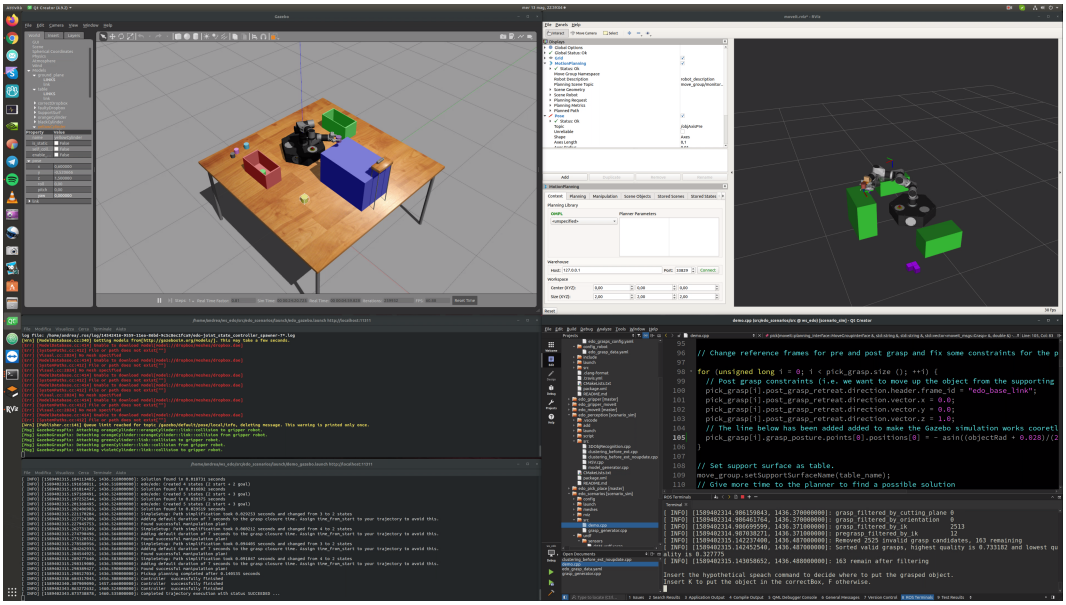

Figure 6.6: Third object Pick.

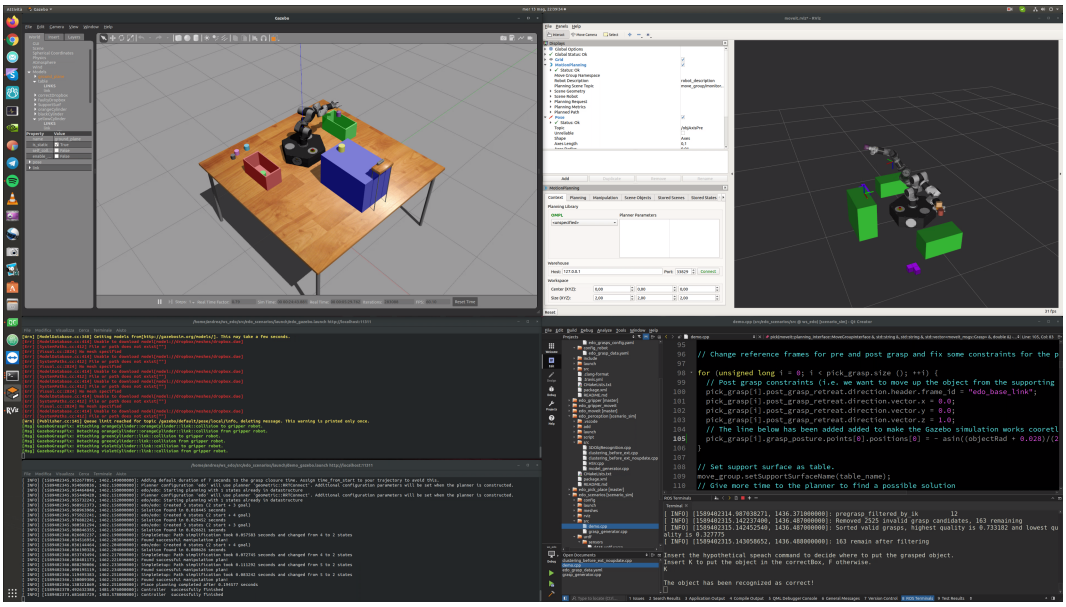

Figure 6.7: Third object Place.

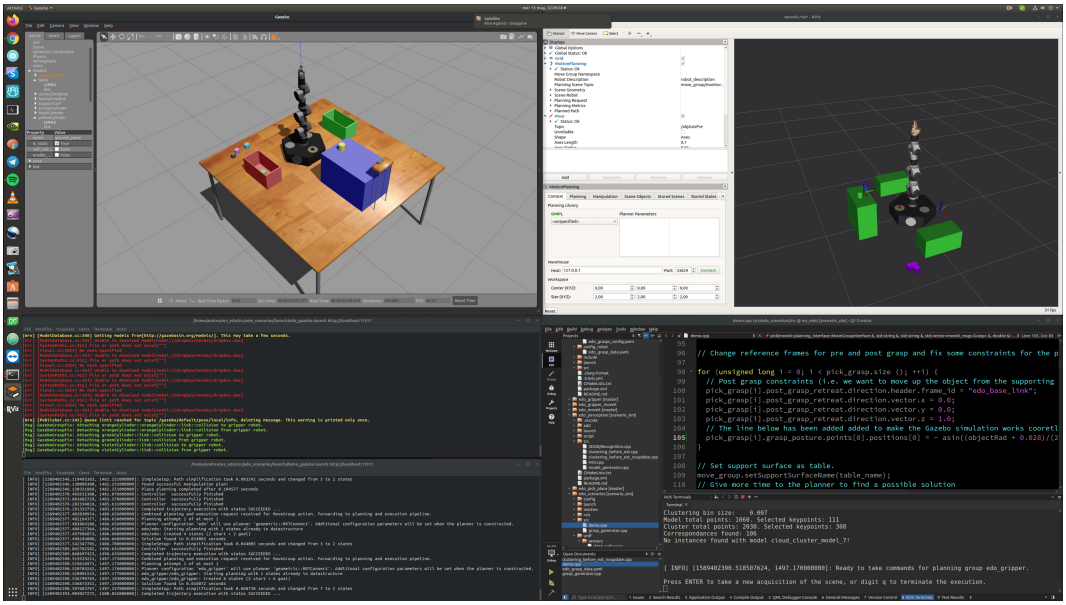

Figure 6.8: Simulation end: The robot moves back to its rest position. New objects are sensed and added to the planning scene, waiting for a new computation.

Chapter 7

Conclusions

The aim of this work is the implementation of a demonstrator scenario related to the Command Recognition in Smart Industry Application use case of the ALOHA project.

The main results achieved in this thesis are:

- Integration of 3D stereocamera:
  - Pointcloud preprocessing, including noise and useless data removal (i.e. support plane);
  - Polynomial pointcloud reconstruction;
  - Pointcloud clusterization;
  - 3D cluster matching with predefined models;
  - Cluster object color and shape identification;
  - Object feature's extraction from clusters.
- Computation of grasps candidates for target objects:
  - Grasp generation, given object's characteristics (i.e. $[x_{depth}, y_{width}, z_{height}]$). Grasp configuration searches for:
    - Grasps with respect to y axis with parallel gripper (X-Z plane);
    - Grasps with respect to x axis with parallel gripper (Y-Z plane);
    - Grasps on faces of object bounding box;
    - Grasps on edges of object bounding box;
  - Grasp filtering. Reduction of possible grasps, removing not feasible solutions due to grasp IK, grasp collision check, grasp cutting plane, grasp orientation, pre-grasp IK and pre-grasp collision check.

- Pick&Place pipeline using 3D sensed objects and grasp generator:
  - Management of dynamic objects in the planning scene, through 3D perception;
  - Grasps computation and movegroup actuation;
  - Management of target object according to the input command.
- Simulation through Gazebo :
  - Introduction of physical aspects, as the presence of gravity and friction between target object and gripper;
  - Highlighted the constraints on the color identification due to the presence of reflective surface for target objects;
  - Highlighted the constraints imposed by the presence of a position controller for the gripper fingers.

Some possible future applications regards the implementation of improvements to brush up the performances:

- Perception: In this step is it possible to define some criteria to identify as interest objects not only cylinders. To do so it would be enough to introduce new models for 3D object recognition, but the tricky steps would be the identification of the position and orientation in space of these objects and their characteristics. Another possible improvement could be the implementation of an algorithm stronger with respect to the light conditions and objects occlusion;
- Demo: Using the input command *follow*. it would be interesting to develop a dynamic pick place system that, respecting the rules of KWS input command recognition, would be able to manage moving objects in space;
- Simulation: To make the simulation much more realistic it would interesting to switch the robot simulation controllers from *position_controllers*, to *effort_controllers*, since the real gripper takes into consideration the constraints imposed by a grasping object, allowing to neglect the opening width of the gripper during pick.

Appendix A

Toolflow setup guide for Smart Industry use case

A.1 Main Steps

This project implements the containerised architecture with all the tool, the DSE engine, the mongoDB, the RedisDB.

1. Clone this project;
2. Follow the steps in the following paragraphs.

A.1.1 Docker_arch

Clone all the other needed projects:

- ***make clone***

the clone target also creates a folder that will contain the data shared among the tools. To know the local repository status of all the tools run:

- ***make status***

Before building the docker architecture, edit the parameters in the file *docker_arch/.env* for your installation, if needed:

- ***docker-compose build***

The first time you build the architecture it can take a while. Ensure to have enough disk space (50 GB should be ok).
To bring up the whole architecture run:

- ***docker-compose up***

Once started it is possible to connecct to the MongoDB and RedisDB through the ports chosed. Use a toolkit like robo3T to browse the MongoDB (https://robomongo.org/download). Use:

- ***redis-cli***

to browse the RedisDB.
To list all the containers run:

- ***docker container ls***: it will show IDs, Names, Ports, . . . ;
- ***docker stop $(docker ps -a -q)***: it forces stop of all the containers;
- ***docker logs container_name -f***: it prints the stdout of a container;
- ***docker exec -it name_of_the_container /bin/bash***: it logins into a container using bash;

A.1.2 Orchestrator

Edit the file *docker_arch/orchestrator-frontend/src/configuration/config.js*, with thee URL to the orchestrator Backend API. For example, if your backend is listening in at the port 5000 in the localhost:

- export const ALOHA_FRONTEND_CONFIGURATION = {
  orchestratorAPIUrl: 'http://localhost:5000/',
  orchestratorWSUrl: 'http://localhost:5000/',
  };

NOTE: The port must be the same chosen in *docker_arch/.env*.
Edit the file *docker_arch/orchestrator-frontend/.env* with the public port for the Orchestrator Frontend. For example if you want the frontend webserver listening at the port 80, add to the *.env* file the line *PORT=80*.
The backend and the frontend ports must be different.
The orchestrator backend starts when you bring up the docker architecture.

A.1.3 Webserver (orchestrator frontend) installation

N.B.: Before launching the frontend you need to install NodeJS 10.13 or higher and npm. Using Ubuntu:

- ***curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -***;

- ***sudo apt-get install -y nodejs***;
- ***sudo apt-get install npm***.

Take a look to: `https://github.com/nodesource/distributions/blob/master/README.md#installation-instructions`

After the installation of NodeJS and npm, install the frontend's dependencies:

- ***cd orchestrator-frontend/***;
- ***npm install***;
- ***cd ..***

A.1.4 Run the webserver

To launch the orchestrator frontend run:

- ***sudo make webserver***

sudo is required if you want to use port 80 or any port under the 1000. If you select the port i.e. 8088 you can launch the frontend without root privileges.

Appendix B

Kinematics

B.1 DH convention

It is a systematic general method to define the relative pose and orientation of two consecutive links. With reference to Figure B.1

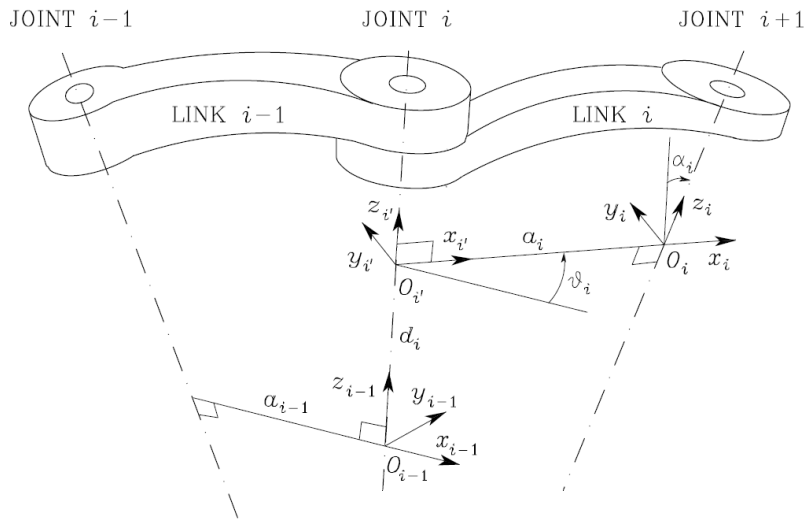

Figure B.1: Denavit-Hartemberg kinematic parameters

let axis i to denote the axis of the joint connecting link *i-1* to link *i*. The so called *Denavit-Hartemberg convention* is adopted to define frame link *i*:

- Choose axis z_i along the axis of Joint *i-1*;

- Locate the origin $\boldsymbol{O}_i$ at the intersection of axis z_i with the common normal to axes z_{i-1} and z_i. Also, locate $\boldsymbol{O}_i$' at the intersection of the common normal with axis z_{i-1};
- Choose axis x_i along the common normal to axes z_{i-1} and z_i with direction from Joint i to Joint $i+1$;
- Choose axis y_i so as to complete a right-handed frame.

The Denavit–Hartenberg convention gives a non-unique definition of the link frame in the following cases:

- For Frame 0, only the direction of axis z_0 is specified, then O_0 and x_0 can be arbitrarily chosen;
- For Frame n, since there is no Joint $n+1$, z_n is not uniquely defined while x_n has to be normal to axis z_{n-1}. Typically, Joint n is revolute, and thus z_n is to be aligned with the direction of z_{n-1};
- When two consecutive axes are parallel, the common normal between them is not uniquely defined;
- When two consecutive axes intersect, the direction of x_i is arbitrary;
- When Joint i is prismatic, the direction of z_{i-1} is arbitrary.

In all such cases, the indeterminacy can be exploited to simplify the procedure; for instance, the axes of consecutive frames can be made parallel.

Once the link frames have been established, the position and orientation of Frame i with respect to Frame $i-1$ are completely specified by the following parameters:

- $a_i \rightarrow$ Distance between $\boldsymbol{O_i}$ and $\boldsymbol{O_i}$';
- $d_i \rightarrow$ Coordinate of $\boldsymbol{O_i}$' along $\boldsymbol{z_{i-1}}$;
- $\alpha_i \rightarrow$ Angle between axis $\boldsymbol{z_{i-1}}$ and $\boldsymbol{z_i}$ about axis $\boldsymbol{x_i}$ (positive for counter-clockwise rotations);
- $\theta_i \rightarrow$ Angle between axis $\boldsymbol{x_{i-1}}$ and $\boldsymbol{x_i}$ about axis $\boldsymbol{z_{i-1}}$ (positive for counter-clockwise rotations).

Two of the four parameters (a_i and α_i) are always constant and depend only on the geometry of connection between consecutive joints established by Link i. Of the remaining two parameters, only one is variable depending on the type of joint that connects Link $i-1$ to Link i. In particular:

- If Joint i is revolute the variable is θ_i;
- If Joint i is prismatic the variable is d_i.
- Choose a frame aligned with Frame *i-1*;
- Translate the chosen frame by d_i along axis $z_{i\text{-}1}$ and rotate it by θ_i about axis $z_{i\text{-}1}$. This sequence aligns the current frame with Frame i' and is described by the homogeneous transformation matrix

$$\boldsymbol{A}_{i'}^{i-1} = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Translate the frame aligned with Frame i' by a_i along axis x_i' and rotate it by α_i about axis x_i'. This sequence aligns the current frame with Frame i and is described by the homogeneous transformation matrix

$$\boldsymbol{A}_{i}^{i'} = \begin{pmatrix} 1 & 0 & 0 & a_i \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- The resulting coordinate transformation is obtained by post-multiplication of the single transformations as

$$\boldsymbol{A}_{i}^{i-1} = \boldsymbol{A}_{i'}^{i-1}\boldsymbol{A}_{i}^{i'} = \begin{pmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Notice that the transformation matrix from Frame $_{\text{i}}$ to Frame $_{\text{i-1}}$ is a function only of the joint variable q_i, that is, θ_i for a revolute joint or d_i for a prismatic joint.

To summarize, the Denavit–Hartenberg convention allows the construction of the direct kinematics function by composition of the individual coordinate transformations expressed in terms of an homogeneous transformation matrix. The procedure can be applied to any open kinematic chain and can be easily rewritten in an operating form as follows.

1. Find and number consecutively the joint axes. Set the directions of axes z_0, $\ldots$, z_{n-1};
2. Choose Frame 0 by locating the origin on axis z_0; axes x_0 and y_0 are chosen so as to obtain a right-handed frame. If feasible, it is worth choosing Frame 0 to coincide with the base frame;

   Execute steps from **3** to **5** for i=1, $\ldots$, n-1:
3. Locate the origin $\boldsymbol{O}_i$ at the intersection of z_i with the common normal to axes z_{i-1} and z_i. If axes z_{i-1} and z_i are parallel and Joint i is revolute, then locate $\boldsymbol{O}_i$ so that $d_i = 0$. If Joint i is prismatic, locate $\boldsymbol{O}_i$ at a reference position for the joint range, e.g., a mechanical limit;
4. Choose axis x_i along the common normal to axes z_{i-1} and z_i with direction from Joint i to Joint $_{i+1}$;
5. Choose axis y_{i-1} so as to obtain a right-handed frame.

   To complete:
6. Choose Frame n. If Joint n is revolute, then align z_n with z_{n-1}, otherwise, if Joint n is prismatic, then choose z_n arbitrarily. Axis x_n is set according to step 4;
7. For $i = 1, \ldots, n$, form the table of parameters $a_i, d_i, \alpha_i, \theta_i$;
8. On the basis of the parameters in 7, compute the homogeneous transformation matrices $\boldsymbol{A}_i^{i-1}$ for $i = 1, \ldots, n$
9. Compute the homogeneous transformation $\boldsymbol{T}_n^0(q) = \boldsymbol{A}_1^0 \ldots \boldsymbol{A}_n^{n-1}$ that yields the position and orientation of Frame n with respect to Frame 0;
10. Given $\boldsymbol{T}_0^b$ and $\boldsymbol{T}_e^n$, compute the direct kinematics function as $\boldsymbol{T}_e^b(q) = \boldsymbol{T}_0^b \boldsymbol{T}_n^0 \boldsymbol{T}_e^n$ that yields the position and orientation of the end-effector frame with respect to the base frame[17].

B.2 Inverse kinematics computation

In this section, all the mathematical computation for the inverse kinematics problem is explained.

Starting from result of direct kinematics problem

$$p_{Wx} = c_1(L_2c_{23} + L_1c_2)$$

$$p_{Wy} = s_1(L_2c_{23} + L_1c_2)$$

$$p_{Wz} = L_2s_{23} + L_1s_2$$

the inverse kinematics problem is solved through the following steps:

- θ_3

$$\begin{aligned} p_{Wx}^2 + p_{Wy}^2 + p_{Wz}^2 &= L_2^2 + L_1^2 + 2L_1L_2(c_2(c_2c_3 - s_2s_3) + s_2(c_2s_3 + s_2c_3)) \\ &= L_2^2 + L_1^2 + 2L_1L_2c_3 \end{aligned}$$

obtaining the formulation for c_3 as

$$c_3 = \frac{p_{Wx}^2 + p_{Wy}^2 + p_{Wz}^2 - L_1^2 - L_2^2}{2L_1L_2}$$

Recalling that the $\cos(\theta)$ function has validity only in the interval [-1, 1], it is easy to say that the *wrist* point of the robot is within the reachable workspace of the manipulator if B.1 is respected:

$$\begin{aligned} &-1 \leq c_3 \leq 1 \Rightarrow -1 \leq \frac{p_{Wx}^2 + p_{Wy}^2 + p_{Wz}^2 - L_1^2 - L_2^2}{2L_1L_2} \leq 1 \\ &\Rightarrow L_1^2 + L_2^2 - 2L_1L_2 \leq p_{Wx}^2 + p_{Wy}^2 + p_{Wz}^2 \leq L_1^2 + L_2^2 + 2L_1L_2 \end{aligned}$$

obtaining finally

$$|L_1 - L_2| \leq \sqrt{p_{Wx}^2 + p_{Wy}^2 + p_{Wz}^2} \leq |L_1 + L_2| \tag{B.1}$$

Recalling that $s_3 = \pm\sqrt{1 - c_3^2}$, the following two solutions are obtained:

$$\theta_3 = atan2(s_3, c_3) = \begin{cases} \theta_{3,I} \in [-\pi, \pi] \\ \theta_{3,II} = -\theta_{3,I} \end{cases}$$

- θ_2

$$\begin{cases} p_{Wx}^2 + p_{Wy}^2 = (L_1 c_2 + L_2 c_{23})^2 \\ p_{Wz} = L_2 s_{23} + L_1 s_2 \end{cases}$$

After obvious mathematical computation, the following result is obtained:

$$\begin{cases} \pm\sqrt{p_{Wx}^2 + p_{Wy}^2} = c_2(L_1 + L_2 c_3) - L_2 s_2 s_3 \\ s_2 = \frac{p_{Wz} - L_2 c_2 s_3}{L_1 + L_2 c_3} \end{cases}$$

that, substituting s_2 in the first equation, finally leads to:

$$\begin{cases} c_2 = \frac{\pm\sqrt{p_{Wx}^2+p_{Wy}^2}(L_1+L_2c_3)+p_{Wz}L_2s_3}{L_1^2+L_2^2+2L_1L_1c_3} \\ s_2 = \frac{p_{Wz}(L_1+L_2c_3)\mp\sqrt{p_{Wx}^2+p_{Wy}^2}L_2s_3}{L_1^2+L_2^2+2L_1L_1c_3} \end{cases}$$

So $\theta_2 = atan2(s_2, c_2)$ and according to the sign of s_3, four possible solution are available, two for $s_3^+ = +\sqrt{1-c_3^2}$ and two for $s_3^- = -\sqrt{1-c_3^2}$

- $\theta 1$

$$\begin{cases} \sqrt{p_{Wx}^2 + p_{Wy}^2} = L_1 c_2 + L_2 c_{23} \\ p_{Wx} = c_1(L_1 c_2 + L_2 c_{23}) \\ p_{Wy} = s_1(L_1 c_2 + L_2 c_{23}) \end{cases}$$

from which

$$\begin{cases} c_1 = \frac{p_{Wx}}{\pm\sqrt{p_{Wx}^2+p_{Wy}^2}} \\ s_1 = \frac{p_{Wy}}{\pm\sqrt{p_{Wx}^2+p_{Wy}^2}} \end{cases}$$

that lead to two different solutions

$$\begin{cases} \theta_{1,I} & = atan2(p_{Wy}, p_{Wx}) \\ \theta_{1,II} & = atan2(-p_{Wy}, -p_{Wx}) \end{cases}$$

Since $atan2(-y,-x) = -atan2(y,-x) = \begin{cases} \pi - atan2(y,x) & if\ y \geq 0 \\ -\pi - atan2(y,x) & if\ y < 0 \end{cases}$

the final result is

$$\theta_{1,II} = \begin{cases} atan2(p_{Wy}, p_{Wx}) - \pi & if\ p_{Wy} \geq 0 \\ atan2(p_{Wy}, p_{Wx}) + \pi & if\ p_{Wy} < 0 \end{cases}$$

It is important to notice that for $p_{Wx} = p_{Wy} = 0$ the inverse kinematics problem does not admit a solution and the robot is in a *SINGULARITY* configuration.

B.3 *ZYZ Euler Angles*

The rotation described by ***ZYZ*** *Euler Angles* is based on the composition of three elementary rotations shown below:

- A first rotation of the origin triad of an angle φ about Z axis, defined by the rotation matrix

$$\boldsymbol{R_z(\varphi)} = \begin{pmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- A second rotation of the new triad of an angle θ about Y axis, defined by the rotation matrix

$$\boldsymbol{R_{y'}(\theta)} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

- A third and final rotation of the new triad of an angle ψ about Z axis, defined by the rotation matrix

$$\boldsymbol{R_{z''}(\psi)} = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

as shown in B.2.

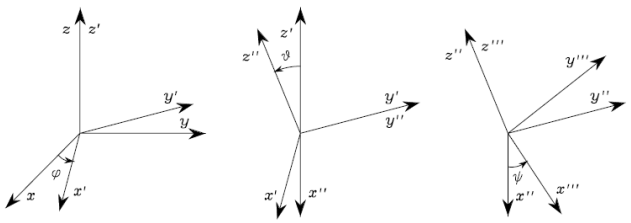

Figure B.2: Representation of ZYZ Euler Angles

Concatenating all the rotations, the following result is obtained

$$\boldsymbol{R} = \boldsymbol{R_z}(\varphi)\boldsymbol{R_{y'}}(\theta)\boldsymbol{R_{z''}}(\psi) = \begin{pmatrix} c\varphi c_\theta c_\psi - s\varphi s_\psi & -c\varphi c_\theta s_\psi - s\varphi c_\psi & c_\varphi c_\theta \\ s\varphi c_\theta c_\psi + c\varphi s_\psi & -s\varphi c_\theta s_\psi + c\varphi c_\psi & s_\varphi s_\theta \\ -s_\theta c_\psi & s_\theta s_\psi & c_\theta \end{pmatrix}$$

To solve the inverse kinematics problem, start from the complete rotation matrix

$$\boldsymbol{R} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

and compute the angles $\varphi, \theta and \psi$ as follow:

- φ Looking at the elements [1, 3] and [2, 3], in the hypothesis that $r_{13} \neq 0$ and $r_{23} \neq 0$, it results:
$$\varphi = atan2(r_{23}, r_{13})$$
- θ Looking at the elements [1, 3] and [2, 3], squaring and adding these two elements and using the element [3, 3], it results:
$$\theta = atan2(\sqrt{r_{13}^2 + r_{23}^2}, r_{33})$$
- ψ Choosing the positive sign for the term $\sqrt{r_{13}^2 + r_{23}^2}$, narrows the membership range to (0, π). Considering this assumption and considering the elements [3, 1] and [3, 2], it results:
$$\psi = atan2(r_{32}, -r_{31})$$

Finally, two different solutions are available, according to the sign of $\sqrt{r_{13}^2 + r_{23}^2}$

- positive sign for $\sqrt{r_{13}^2 + r_{23}^2}$:
$$\begin{cases} \varphi = atan2(r_{23}, r_{13}) \\ \theta = atan2(\sqrt{r_{13}^2 + r_{23}^2}, r_{33}) \\ \psi = atan2(r_{32}, -r_{31}) \end{cases}$$
- negative sign for $\sqrt{r_{13}^2 + r_{23}^2}$:
$$\begin{cases} \varphi = atan2(-r_{23}, -r_{13}) \\ \theta = atan2(-\sqrt{r_{13}^2 + r_{23}^2}, r_{33}) \\ \psi = atan2(-r_{32}, r_{31}) \end{cases}$$

These two solutions degenerate for $s_\theta = 0$. In this case it is possible to determine only the sum or the difference of φ and ψ. In fact, if $\theta = 0, \pi$, the successive rotations of φ and ψ are made around axis triads that are parallel to each other, giving the same contribution to the complete rotation[17].

B.4 Anthropomorphic arm configurations

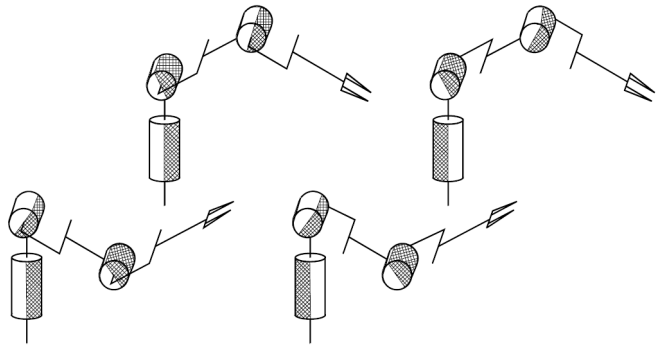

Figure B.3: The four configurations of an anthropomorphic arm compatible with a given wrist position

According to the results obtained by 2.3, 2.4 and 2.5, it is evident that four possible solutions are available:

- $(\theta_{1,I}, \theta_{2,I}, \theta_{3,I})$
- $(\theta_{1,I}, \theta_{2,III}, \theta_{3,II})$
- $(\theta_{1,II}, \theta_{2,II}, \theta_{3,I})$
- $(\theta_{1,II}, \theta_{2,IV}, \theta_{3,II})$

According to B.3, the four possible solutions are respectively:

- Shoulder-right/Elbow-up
- Shoulder-left/Elbow-up
- Shoulder-right/Elbow-down
- Shoulder-left/Elbow-down

It is important to notice that it is possible to find the solutions only if at least:

$$p_{Wx} \neq 0 \quad OR \quad p_{Wy} \neq 0$$

In the case $p_{Wx} = p_{Wy} = 0$, an infinity of solutions is obtained, since it is possible to determine the joint variables θ_2 and θ_3 independently of the value of θ_1. In such configuration the arm is kinematically *singular*[17].

B.5 Detailed steps for e.DO connection

Firstly start the cobot using the external switch on the base. After few seconds a WiFi net will become available (*edo.wifi.3d:ca:23*). The automatic boot of the control software has been disabled, so it is necessary to link the cobot with a laptop or with the dedicated app, to start the control software. When available, it is preferable to use the wired connection. Anyway, to use the two available connections, follows this procedure:

- Wired connection Connect the Ethernet wire to the laptop and configure manually the *IP* address of this connection. This step is necessary since no *DHCP* is available.

  To configure the wired connection, set:

  – IP: 10.42.0.1

  – Mask: /24

  The robot will become available at the IP address 10.42.0.49.

- Connect the laptop to the WiFi net *edo.wifi.3d:ca:23* and insert the password *edoedoedo*. Since in this case the *DHCP* is available, an IP address will be assigned to the laptop. The robot will become available at the IP address 192.168.12.1.

Once connected to one of the two available nets, it is possible to connect to the robot through *SSH* using, on a new shell, *ssh edo@<Robot IP>* and *raspberry* as password.

To calibrate the robot or to check easily the status of the brakes, use the dedicated app, following the procedure described in it. The serial number to which the app refers at the beginning of the procedure is the one that starts with *2018-...* on the base of the robot.

Appendix C

Perception Pipeline

C.1 Code Snippets

C.1.1 Code snippet of the addCylinder() function

```
void addCylinder()
  {
    double thr = 0.005;
    std::string cyl = "cylinder";
    std::string cyl_name;
    moveit::planning_interface::PlanningSceneInterface
                                planning_scene_interface;
    // Adding Cylinder to Planning Scene
    // ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    // Define a collision object ROS message.
    moveit_msgs::CollisionObject collision_object;
    collision_object.header.frame_id
    = "camera_depth_optical_frame";

    //Skip the first cycle
    if (skip_first == false) {
      for (size_t k = 0; k < prev_names.size(); k++) {
        //Check if the new obj_i is the same as one of the previous
        //estimation
        if (abs(abs(cylinder_params.height)
                -abs(previous_computations[h_cnt]))<thr){
          r_cnt = h_cnt + 1;
          if (abs(abs(cylinder_params.radius)
              -abs(previous_computations[r_cnt]))<thr){
```

```
        CoG_cnt = h_cnt - 3;
        find = true;
        //Remove previous object if the cylinder CoG is
        //different within a tolerance of 5 mm!
        std::cout << "Estimation uncertanty wrt previous
                     estimation: "<< '\n';
        for (size_t i = CoG_cnt; i < CoG_cnt + 3; i++) {
          std::cout << abs(abs(cylinder_params.center_pt[i])
                     -abs(previous_computations[i])) << '\n';
          if (abs(abs(cylinder_params.center_pt[i])
             -abs(previous_computations[i])) > thr
             && rmv_obj == false) {
            rmv_obj = true;
            if (rmv_obj == true) {
              new_names.push_back(prev_names[prev_name_index]);
              std::cout << "\n";
              ROS_INFO("Removing object from the world...");
              collision_object.id = new_names.back();
              collision_object.operation
              =collision_object.REMOVE;
              planning_scene_interface.
              applyCollisionObject(collision_object);
            }
          }
        }
        if (rmv_obj == false){
          new_names.push_back(prev_names[prev_name_index]);
          std::cout << "The object " << new_names.back() <<
                       " has not changed!" << '\n';
        }
      }
    }
    else {
      h_cnt = h_cnt + 5;
    }
    prev_name_index++;

    std::cout << "\n\n" << '\n';
  }

  CoG_cnt = 0;
```

```
    h_cnt = 3;
    r_cnt = 4;

    if (find == false) {
      new_obj_cnt = new_obj_cnt + prev_names.size();
      new_names.push_back(
      "cylinder" + std::to_string(new_obj_cnt));
      new_obj = true;
    }

    std::cout << "\n\n" << '\n';
  }

  if (skip_first == true || rmv_obj == true || new_obj == true) {
    //Save previous values of the CoG, Height and Radius to make
    //the comparison with the new estimation
    //CoG
    previous_computations.push_back(cylinder_params.center_pt[0]);
    previous_computations.push_back(cylinder_params.center_pt[1]);
    previous_computations.push_back(cylinder_params.center_pt[2]);
    //Height
    previous_computations.push_back(cylinder_params.height);
    //Radius
    previous_computations.push_back(cylinder_params.radius);

    if (skip_first == true) {
      cyl_name = cyl + std::to_string(cyl_cnt);
      collision_object.id = cyl_name;
      //Save first names
      new_names.push_back(cyl_name);
    }
    else if (new_obj == true) {
      collision_object.id = new_names.back();
    }
    else {
      collision_object.id = prev_names[prev_name_index-1];
      //Save previous names
      new_names.push_back(prev_names[prev_name_index-1]);
    }

    // Define a cylinder which will be added to the world.
```

```
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.CYLINDER;
primitive.dimensions.resize(2);
/* Setting height of cylinder. */
primitive.dimensions[0] = cylinder_params.height;
/* Setting radius of cylinder. */
primitive.dimensions[1] = cylinder_params.radius;

// Define a pose for the cylinder
//(specified relative to frame_id).
geometry_msgs::Pose cylinder_pose;
//Computing and setting quaternion
//from axis angle representation.
Eigen::Vector3d cylinder_z_direction(
      cylinder_params.direction_vec[0],
      cylinder_params.direction_vec[1],
      cylinder_params.direction_vec[2]);
Eigen::Vector3d origin_z_direction(0., 0., 1.);
Eigen::Vector3d axis;
axis = origin_z_direction.cross(cylinder_z_direction);
axis.normalize();
double angle=acos(
             cylinder_z_direction.dot(origin_z_direction));
cylinder_pose.orientation.x = axis.x() * sin(angle / 2 +
                                             cam_roll / 2);
cylinder_pose.orientation.y = axis.y() * sin(angle / 2 +
                                             cam_pitch / 2);
cylinder_pose.orientation.z = axis.z() * sin(angle / 2 +
                                             cam_yaw / 2);
cylinder_pose.orientation.w = cos(angle / 2);

// Setting the position of cylinder.
cylinder_pose.position.x = cylinder_params.center_pt[0];
cylinder_pose.position.y = cylinder_params.center_pt[1];
cylinder_pose.position.z = cylinder_params.center_pt[2];

// Add cylinder as collision object
collision_object.primitives.push_back(primitive);
collision_object.primitive_poses.push_back(cylinder_pose);
collision_object.operation = collision_object.ADD;
planning_scene_interface.applyCollisionObject(
```

```
                              collision_object);
    if (skip_first == true) {
      std::cout << "\nCollision object "<< cyl_name
                << " has been added.\n\n";
    }
    else if (new_obj == true) {
      std::cout << "\nCollision object "<< new_names.back()
                << " has been added.\n\n";
    }
    else {
      std::cout << "\nCollision object "
                << new_names[new_name_index]
                << " has been added.\n\n";
    }
    new_name_index++;
  }
}
```

C.1.2 Code snippet of the Update Planning Scene Control Sequence

```
  ROS_INFO("Updating the planning scene...");
      //Remove all the elements that are not
      //present in the scene anymore
      for (size_t i = 0; i < prev_names.size(); i++) {
        for (size_t j = 0; j < obj_index.size(); j++){
          if(prev_names[i] == "cylinder" +
                              std::to_string(obj_index[j])){
            ext_obj = true;
          }
        }
        if (ext_obj == false){
          ROS_INFO("Removing objects that are
                  not present in the scene anymore...");
          previous_computations.erase(
                               previous_computations.begin()
                               + 5*i,
                               previous_computations.begin()
                               + 5*i + 5);
          moveit::planning_interface::PlanningSceneInterface
                                      planning_scene_interface;
```

```
        moveit_msgs::CollisionObject collision_object;
        collision_object.header.frame_id
        = "camera_depth_optical_frame";
        collision_object.id = prev_names[i];
        collision_object.operation = collision_object.REMOVE;
        planning_scene_interface.applyCollisionObject(
                                  collision_object);
      }
      ext_obj = false;
    }
```

C.2 Quaternions and Pose Estimation

Firstly let's introduce the key concepts for the object orientation identification.

C.2.1 Axis Angle representation

It is a non minimum representation of the object orientation, starting from four parameters, with express the rotation of a certain angle around a rotation axis in space.

With reference to Figure C.1, the rotation matrix of $\mathbf{R}(\theta,\mathbf{r})$, that represents the rotation of an angle θ around the axis $\mathbf{r}$, can be expressed as a composition of elementary rotations with respect to the axis of the reference triad as follow:

1. Align $\mathbf{r}$ with z, which is obtained as the sequence of a rotation by $-\alpha$ about z and a rotation by $-\beta$ about y;
2. Rotate by θ about z;
3. Realign with the initial direction of $\boldsymbol{r}$, which is obtained as the sequence of a rotation by β about y and a rotation by α about z.

The resulting rotation matrix is:

$$\boldsymbol{R}(\theta,\boldsymbol{r}) = \boldsymbol{R}_z(\alpha)\boldsymbol{R}_y(\beta)\boldsymbol{R}_z(\theta)\boldsymbol{R}_y(-\beta)\boldsymbol{R}_z(-\alpha)$$

Solving the rotations, the following result is achieved:

$$\mathbf{R}(\theta,\boldsymbol{r}) = \begin{bmatrix} r_x^2(1-c_\theta)+c_\theta & r_x r_y(1-c_\theta)-r_z s_\theta & r_x r_z(1-c_\theta)+r_y s_\theta \\ r_x r_y(1-c_\theta)+r_z s_\theta & r_y^2(1-c_\theta)+c_\theta & r_y r_z(1-c_\theta)-r_x s_\theta \\ r_x r_z(1-c_\theta)-r_y s_\theta & r_y r_z(1-c_\theta)+r_x s_\theta & r_z^2(1-c_\theta)+c_\theta \end{bmatrix} \quad \text{(C.1)}$$

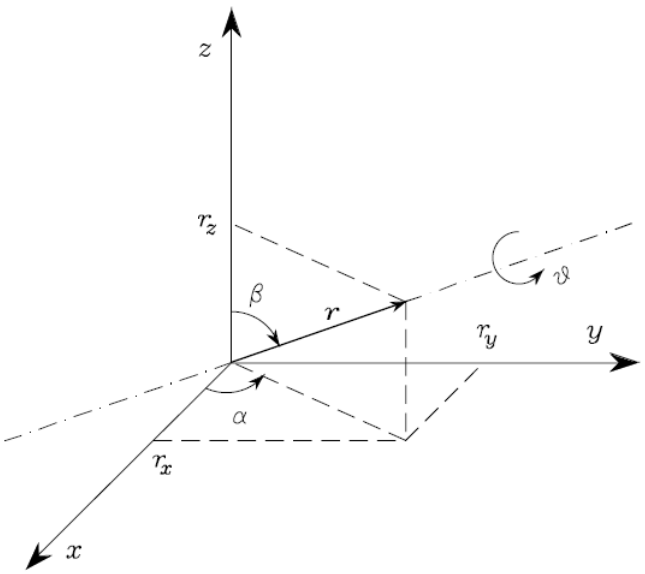

Figure C.1: Rotation of an angle about an axis

for which holds:

$$\boldsymbol{R}(-\theta, -\boldsymbol{r}) = \boldsymbol{R}(\theta, \boldsymbol{r})$$

that involves a *non-unique* representation, since a rotation by θ around $\boldsymbol{r}$ cannot be distinguished with respect to a rotation by $-\theta$ around -$\boldsymbol{r}$.

For more details about *Axis Angle representation* take a look to [17].

C.2.2 Quaternion

It represents the extension of the complex number set, based on a four parameters representation. A *unit quaternion* is the mathematical component defined as $\boldsymbol{Q}(\eta, \boldsymbol{\epsilon})$, where:

$$\eta = \cos\frac{\theta}{2} \quad \textit{is the scalar part}$$

$$\boldsymbol{\epsilon} = \sin\frac{\theta}{2}\boldsymbol{r} \quad \textit{is the vectorial part} \quad \epsilon = [\epsilon_x, \epsilon_y, \epsilon_z]^T$$

On this component, the following relation is always valid:

$$\eta^2 + \epsilon_x^2 + \epsilon_y^2 + \epsilon_z^2 = 1$$

For more details about *Quaternion* take a look to [17].

C.2.3 Pose and Orientation for the Collision Object

For the identification of the *Pose* of the collision object the task is relatively simple, since the coordinates of the center point are provided directly by the *coefficient_cylinder* structure, meanwhile for the identification of the orientation, the task is a little bit complex.

In detail, starting form the direction vector, already estimated by the *extractCylinder* function, the aim is to compute the quaternion components. This task can be performed, following these steps:

1. Cross product between the unit vector along which the object should spawn (in this case the z axis of the camera_infra1_optical_frame) and the direction vector of the cylinder (result saved in a Vector3D called *axis*);

2. Normalization of the vector representing the triad of the collision object;

3. Computation of the angle $\theta = acos(dot\_product\_of\_the\_two\_vectors)$. It represents the rotation about the direction vector;

4. Definition of the collision object orientation:

   For what concern the object orientation it is necessary to take care of the camera **RPY** configuration wrt the scene, as can be seen in Figure C.2. Practically, considering the non-null cam_roll angle, an orientation compensation for the x component of the quaternion is necessary. This consideration is valid for all the components of the quaternion and for convention, the sign is chosen positive if the rotation is counterclockwise and negative otherwise, obtaining:

   - $orientation.x = axis.x() * \sin(\frac{\theta}{2} + \frac{cam_{roll}}{2})$;
   - $orientation.y = axis.y() * \sin(\frac{\theta}{2} + \frac{cam_{pitch}}{2})$;
   - $orientation.z = axis.z() * \sin(\frac{\theta}{2} + \frac{cam_{yaw}}{2})$;
   - $orientation.w = \cos\frac{\theta}{2}$.

The code snippet that computes these steps is shown below:

```
// Define a pose for the cylinder (specified relative to frame_id).
  geometry_msgs::Pose cylinder_pose;
  //Computing and setting quaternion from axis angle
  //representation.
  Eigen::Vector3d cylinder_z_direction(
            cylinder_params.direction_vec[0],
            cylinder_params.direction_vec[1],
            cylinder_params.direction_vec[2]);
```

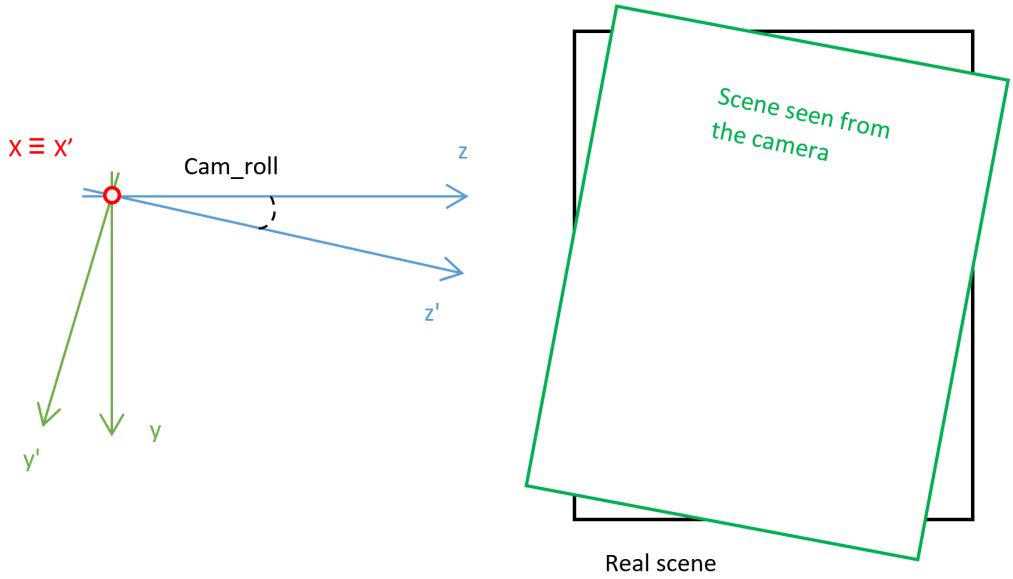

Figure C.2: RPY cam configuration wrt scene

```
Eigen::Vector3d origin_z_direction(0., 0., 1.);
Eigen::Vector3d axis;
axis = origin_z_direction.cross(cylinder_z_direction);
axis.normalize();
double angle = acos(cylinder_z_direction.dot(origin_z_direction));
cylinder_pose.orientation.x = axis.x() * sin(angle / 2 +
                                             cam_roll / 2);
cylinder_pose.orientation.y = axis.y() * sin(angle / 2 +
                                             cam_pitch / 2);
cylinder_pose.orientation.z = axis.z() * sin(angle / 2 +
                                             cam_yaw / 2);
cylinder_pose.orientation.w = cos(angle / 2);
```

Bibliography

[1] URL: https://www.aloha-h2020.eu/project/project-overview# (cit. on pp. ii, 2).

[2] P.Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. Tech. rep. Apr. 2018. URL: https://arxiv.org/pdf/1804.03209.pdf,%20http://download.tensorflow.org/data/speech_commands_v0.02.tar.gz (cit. on p. 5).

[3] URL: https_//github.com/petewarden/open-speech-recording (cit. on p. 5).

[4] URL: https://gitlab.com/aloha.eu/data_standardization/tree/master (cit. on p. 7).

[5] URL: https://www.st.com/en/evaluation-tools/steval-stlkt01v1.html (cit. on p. 8).

[6] URL: https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html (cit. on p. 8).

[7] Xuejiao Li xjli and Zixuan Zhou zixuan. «Speech Command Recognition with Convolutional Neural Network». In: 2017 (cit. on pp. 9, 10, 16).

[8] Tara Sainath and Carolina Parada. «Convolutional Neural Networks for Small-Footprint Keyword Spotting». In: *Interspeech*. 2015 (cit. on pp. 9–11, 16).

[9] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV] (cit. on p. 11).

[10] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. «Hello Edge: Keyword Spotting on Microcontrollers». In: (Nov. 2017) (cit. on pp. 11, 12, 16).

[11] Raphael Tang, Weijie Wang, Zhucheng Tu, and Jimmy Lin. «An Experimental Analysis of the Power Consumption of Convolutional Neural Networks for Keyword Spotting». In: Apr. 2018, pp. 5479–5483. DOI: 10.1109/ICASSP.2018.8461624 (cit. on pp. 11, 16).

[12] URL: https://www.st.com/en/evaluation-tools/steval-stlkt01v1.html (cit. on p. 15).

[13] URL: http://www.openstm32.org/HomePage (cit. on p. 15).

[14] URL: https://putty.org/ (cit. on p. 15).

[15] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. 2018. arXiv: 1804.03209 [cs.CL] (cit. on p. 16).

[16] URL: https://edo.cloud/it/il-robot/ (cit. on p. 17).

[17] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. 1st. Springer Publishing Company, Incorporated, 2008. ISBN: 1846286417 (cit. on pp. 23, 26, 95, 99, 100, 108).

[18] Wikipedia. *Robot Operating System — Wikipedia, L'enciclopedia libera*. [Online; in data 13-marzo-2020]. 2019. URL: http://it.wikipedia.org/w/index.php?title=Robot_Operating_System&oldid=106936622 (cit. on p. 26).

[19] L. Joseph. *Mastering ROS for Robotics Programming*. Packt Publishing, 2015. ISBN: 9781785282997. URL: https://books.google.it/books?id=2jjlCwAAQBAJ (cit. on pp. 27, 28).

[20] URL: http://wiki.ros.org/rviz (cit. on p. 27).

[21] URL: http://moveit.ros.org/ (cit. on p. 27).

[22] Radu Bogdan Rusu and Steve Cousins. «3D is here: Point Cloud Library (PCL)». In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011 (cit. on p. 32).

[23] Wikipedia contributors. *Random sample consensus — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-March-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Random_sample_consensus&oldid=943638739 (cit. on p. 35).

[24] Wikipedia contributors. *K-d tree — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-March-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=K-d_tree&oldid=940106037 (cit. on p. 35).

[25] URL: http://docs.pointclouds.org/trunk/structpcl_1_1_point_x_y_z_r_g_b.html (cit. on p. 44).

[26] URL: http://docs.ros.org/melodic/api/moveit_ros_planning_interface/html/classmoveit_1_1planning__interface_1_1MoveGroupInterface.html (cit. on p. 53).

[27] URL: http://docs.ros.org/melodic/api/moveit_msgs/html/msg/Grasp.html (cit. on p. 53).

[28] URL: http://docs.ros.org/jade/api/moveit_msgs/html/msg/PlaceLocation.html (cit. on pp. 54, 60).

[29] URL: http://docs.ros.org/jade/api/tf2/html/classtf2_1_1Quaternion.html (cit. on p. 57).

[30] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. «The Open Motion Planning Library». In: *IEEE Robotics & Automation Magazine* 19.4 (Dec. 2012). http://ompl.kavrakilab.org, pp. 72–82. DOI: 10.1109/MRA.2012.2205651 (cit. on p. 76).

[31] URL: http://gazebosim.org/ (cit. on p. 82).

[32] URL: https://github.com/jenniferBuehler/gazebo-pkgs/wiki/The-Gazebo-grasp-fix-plugin (cit. on p. 82).