Version: 1
Rev. 1

Y. 2019

UNIVERSITÀ
DEGLI STUDI
DI SALERNO

.DIEM

Cognitive Robotic Project: Object grasping

Members
Di Prisco Giovanni
Allegretti Giovanni
Schettini Marco
Scaldaferri Antonello

Project: Object Grasping

Summary

1. Project Requirements

About used hardware.

1.1 Niryo One

Niryo One is a 6-axis robotic arm. That means 6 degrees of freedom for the robot. As showed in figure 1, the robot can be broken down into 7 parts.

1 - Base	Orange
2 - Shoulder	.Yellow
3 - Arm	.Green
4 - Elbow	.Turquoise
5 - Forearm	.Blue
6 - Wrist	Purple
7 - Hand	Red

Figure 1: Niryo One composition.

Figure 2: Niryo One Robot

There are 3 software layers when you use Niryo One. From higher to lower:

- Niryo One Studio (desktop app)
- Niryo One Raspberry Pi image (robot software) which contains the ROS stack
- Niryo Stepper (motors firmware)

1.2 Nvidia Jetson Nano

NVIDIA® Jetson Nano™ is a small, powerful computer that lets you run multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that runs in as little as 5 watts.

GPU	Maxwell 128 NVIDIA CUDA® cores
CPU	Quad-core ARM® Cortex®-A57
Memory	4 GB 64-bit LPDDR4
Storage	16 GB eMMC 5.1 Flash
Video Encode	4K @ 30 (H.264/H.265)
Video Decode	4K @ 60 (H.264/H.265)

Figure 3: Nvidia Jetson Nano

1.3 Intel Real Sense Depth Camera

The Intel® RealSense™ D435 offers the widest field of view of all our cameras, along with a global shutter on the depth sensor that is ideal for fast moving applications.

The Intel® RealSense™ depth camera D435 is a stereo tracking solution, offering quality depth for a variety of applications. It's wide field of view is perfect for applications such as robotics or augmented and virtual reality, where seeing as much of the scene as possible is vitally important. With a range up to 10m, this small form factor camera can be integrated into any solution with ease and comes complete with our Intel RealSense SDK 2.0 and cross-platform support.

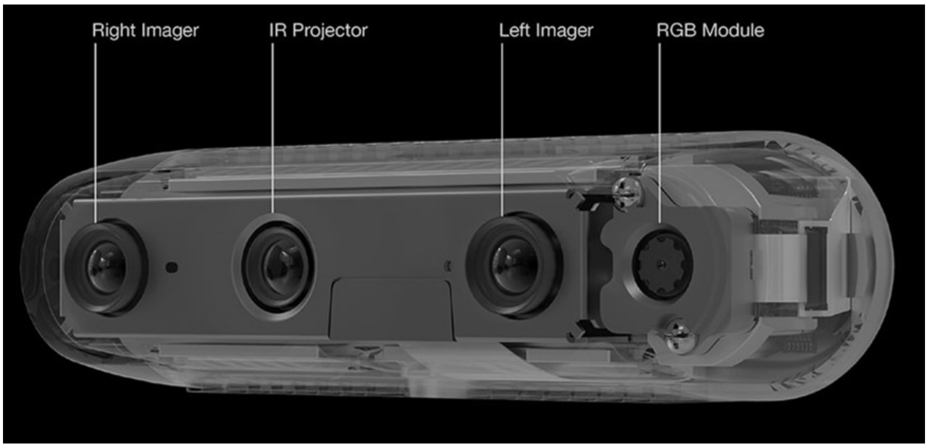


Figure 4: Intel® RealSense™ D435

2. Aim of the project

The purpose of the project is to develop an autonomous system which detect and recognize the highest number of different coloured balls inside a controlled environment. Moreover, in a given time interval, the robot has to grasp every ball that was recognized and put it into the correspondent coloured box.

The equipment:

- N balls (having 4 different colours).
- 4 boxes (having 4 different colours, the same of the balls).

3. Design

In this chapter has been described the hardware and software choices made to reach the goal of the project. About Hardware architecture, we have to describe where locate the camera and how we have made little structural changes on the robot in order to accommodate some constrains that we are going to discuss. On the other hands, the developed functions must integrate with the ROS modules of the framework seamlessly, thus requiring a careful architectural software design.

3.1 Hardware architecture

We decide to use only one Real Sense to reach the goal. The camera is positioned on the 5th joint, in this way we get more precision in a short range giving up the speed obtainable by having a zenithal camera. Since the RealSense has a lower limit on the depth calculation of about 15 cm, an extension has been added to the end effector in order to be able to see at any time.

3.2 Software architecture

The first constraint refers to the ROS framework, it was used because the hardware requires different control point, both Niryo and Jetson are equipped with ROS that come useful when we have to exchange information across different platform end-point. In this way, we can choice where we want to develop and 'silently' debug.

To fulfil the task, we have to carefully choice the network and we must develop various inter-dependent modules that will then be integrated into the final system.

3.2.1 About Network choice

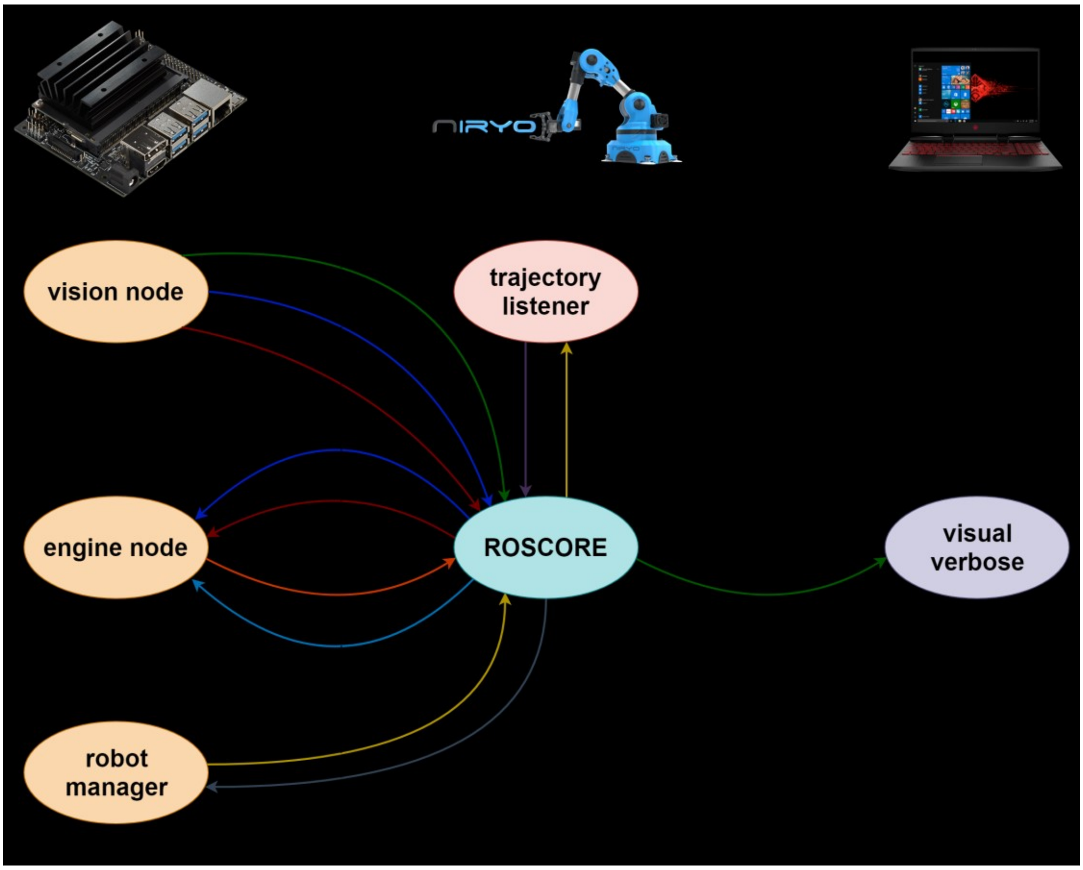
To detect and classify the balls and boxes, it was decided to use a deep neural network based detector and classifier. This is because, today, there are numerous state-of-the-art detectors, which can be easily used in their own applications, such as detectors of customized objects, simply by fine-tuning the network.

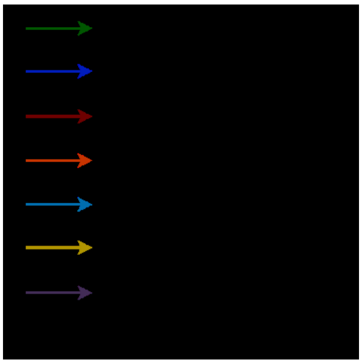
An alternative to this solution could have been the separate use of a detector of generic objects, like, for example, the Viola & Jones algorithm and the subsequent use of a downstream classifier, which, taken as input the generic object detected by the upstream detector, performs the classification of this object in a predefined number of classes, using, for example, classifiers such as Support Vector Classifier, K-Nearest Neighbour, or a clustering algorithm such as K-means.

Instead, it was decided to use a state-of-the-art single shot detector based on deep neural network, because, as mentioned before, we wanted to reuse what was already present in the literature and focus on the system-integration process.

In particular, given its high efficiency on mobile devices, it was decided to use MobileNetSSDv2, which is a detector based on deep neural network, that uses the Single-Shot-Detection methodology for the detection of objects of interest. This network allows for a good compromise between performance and speed.

3.2.2 Node design





4. Implementation

4.1 Vision

4.1.1 Acquiring dataset

To use the detector to detect objects of our interest (balls and boxes of 4 different colours), the network was fine-tuned on a data set we collected.

The data set was collected by capturing images directly using the Real Sense camera, since the video stream will be acquired through the same device during the competition.

The dataset that was acquired is composed of 504 images, of which 441 were used as training set during the fine-tuning process and the other 63 as validation set (approximately 10% of the acquired images). In this data set there are 2304 objects, of which 2062 in the training set and the remaining 242 in the validation set. In particular, there are a total of:

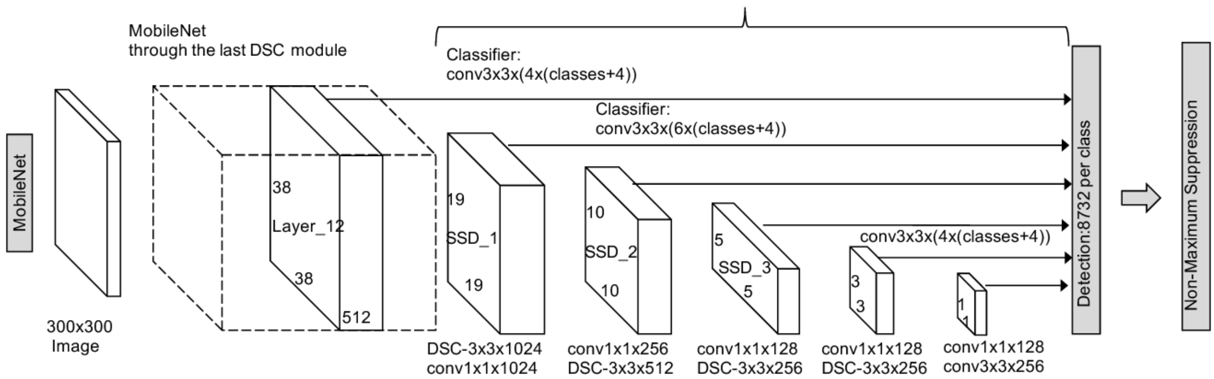
- 520 red balls
- 470 blue balls
- 377 green balls
- 600 white balls
- 90 red boxes
- 79 blue boxes
- 88 green boxes
- 80 white boxes

4.1.2 Detector Details

Since the object detector used is MobileNetSSDv2, this means that the backbone used as feature-extractor is the deep neural network based classifier MobileNetv2. The power of this objects detector is that the same features extracted to perform the classification are also used to perform the detection of the objects in the image. Below is the list of the various layers that make up this feature-extractor:

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

About the Single-Shot-Detection technique, the image is divided into a grid with a certain number of cells, on each cell different portions of the image are considered, at different scales and with different aspect-ratio. Below is an example architecture of deep neural network based detector, which use the Single-Shot-Detection technique:



4.1.3 Fine-Tuning

The fine-tuning of this deep neural network was performed using the Object Detection API of Python's TensorFlow framework. In particular, the training was performed on 9000 iterations and a batch size of 24. So, each epoch is composed of 504/24 iterations = 21 iterations, for a total of 9000/21 ≅ 429 epochs, with a constant learning rate equal to 0.004. Moreover, during the training a random cropping operation was performed for data augmentation.

In this particular case, of the performed fine-tuning, 6 different grid layers were used for which to create the anchors, with a scale factor for the anchors ranging from 0.2 (which corresponds to the finest resolution) to 0.95 (which corresponds to the coarsest resolution). In addition, the following 5 different aspect ratios were used for each cell: 1:1, 2:1, 3:1, 1:2, 1:3.

Finally, the network takes images at 300x300 pixels resolution, then an image resize operation is previously performed, to bring the image back to the desired resolution, inserting a uniform background for images with an aspect ratio different from 1.

4.1.4 Network Output

This network, for each input image, outputs a tensor containing 4 elements:

- The first element indicates the number of objects detected in the image.
- The second element indicates, for each object detected in the image, the "objectness" relative to the detected object (it indicates how likely it is to be an object).
- The third element indicates, for each object detected in the image, the bounding box relative to the detected object (therefore it contains the 4 elements that will indicate the coordinates of the Top-Left and Bottom-Right points of the bounding box).
- The fourth element indicates, for each object detected in the image, the class identifier relative to the detected object.

4.1.5 Vision chain

The task of **vision_node** is to execute the chain of vision from acquisition to classification:

Acquire images -> Recognize objects -> Get relative positions of recognized objects.

First, we acquire the images streaming: coloured and depths images. The first type of image is passed to the net in order to recognize the objects. The second type is passed to a series of function provided by the RealSense SDK, that align the depth image with the coloured. In this way the phase displacement is reduced and the precision with the real values increase.

Image acquisition

When the images acquisition is done, the RGB images are processed with the DNN. This part we collect the output information of the net into an array and then we publish this information on a topic named **net_topic**. At the same time, we use OpenCV library to draw the bounding boxes calculated by the net on the passed image. This image is published on another topic called **image_topic** used only to have a visual feedback.

Image processing

Last, we calculate the cloud points from the depth images aligned to the coloureds. Before it, to achieve the best precision by cloud points, we pre-process the depth image with a series of filters: decimator filter, spatial and temporal filter. Thanks to this pre-processing the values calculated by the camera are less random and the variation is on the millimetre scale. To obtain the right position of the recognized object we extract the centroids of the bounding boxes obtained from the net and we search the corresponding cloud points. In addition, to obtain more robust values and to avoid minimal blockages we decide to calculate the mean values set of cloud points surrounding the centroid. Also, this information is collected into an array and published on a topic named **cloudpoint_topic**

4.2 Engine

4.2.1 Perception

On the first stages the robot has to map all the objects inside the environment.

Angular sectors

As first logic, the environment has to be divided in different angular sector, this choice is made depending on the fact that the camera has an

associated point of view.

when the camera covers the sector, a scan is performed to obtain the position of any objects.

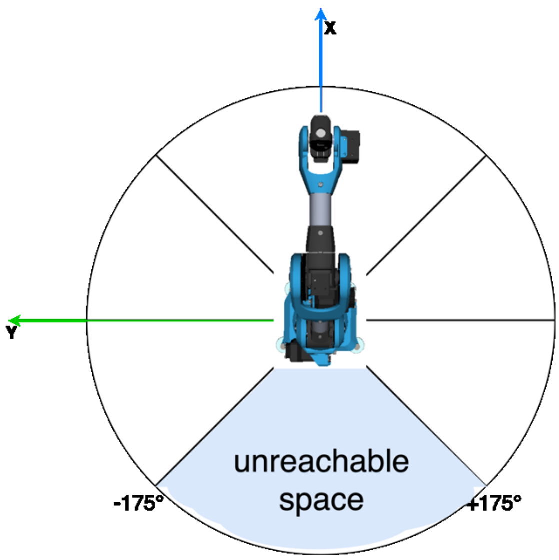


Figure 5: Top view angular sectors

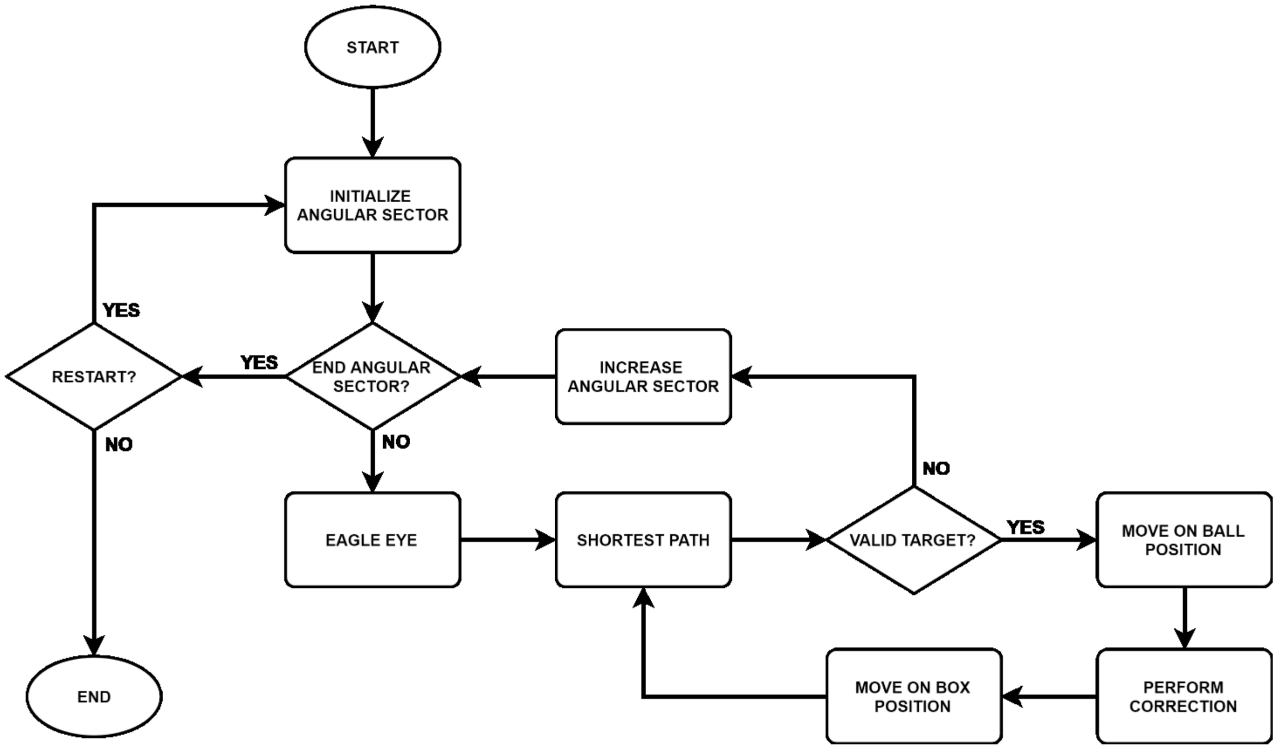


Figure 6: Engine flowchart

Getting object position

Once **Point Cloud** have been obtained, we have to get x, y, z object coordinates from **Real Sense** reference system. It means that we have to rotate and translate a point from the Real Sense reference system (relative position) to the **Niryo One** reference system (absolute position).

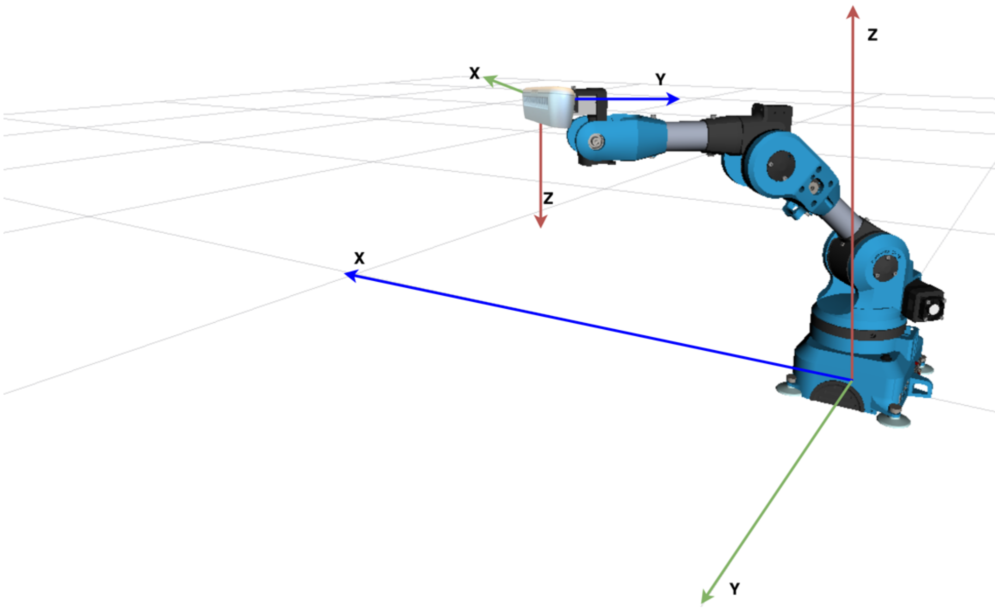


Figure 7: Niryo and Real Sense Coordinates Systems

From the geometrical point of view, it can be reassumed as in Figure 6:

The absolute position of the object (ball/box) is: $\boldsymbol{p^0} = \boldsymbol{o_1^0} + \boldsymbol{R_1^0}\boldsymbol{p^1}$

Where:

- $\boldsymbol{o_1^0}$ is the camera position (end-effector position + offset)
- $\boldsymbol{R_1^0}$ is rotation matrix of **camera** coordinates system respect **niryo one** coordinates system (in our case as rotation angle we've considered the rotation angle of the first joint along the z axis)
- $\boldsymbol{p^1}$ is the object position from camera perspective (obtained from point cloud)

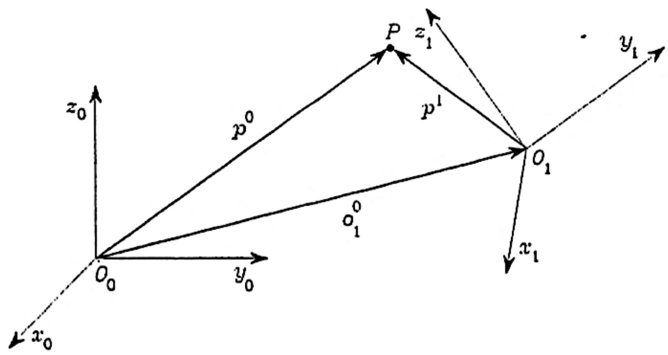


Figure 8: Point representation from different coordinates systems

Note: before applying this transformation, we must rotate Real Sense coordinates system of 90 degrees and invert the Y axis sign in order to have aligned axes in the initial pose (0, 0, 0).

4.2.2 Reasoning

Mapping scene

To ensure robustness and cut down the presence of false positives, the observed objects are validated if their occurrence in a fixed number of frames exceeds a threshold value.

Shortest path

To optimize the robot trajectory and reduce the times we compute the shortest path on the just mapped scene. As shortest path we consider the sum of the Euclidean distance between ball and box plus the Euclidean distance between ball and gripper, sorting in increaser order we obtain the first more convenient target to grasp and deliver.

4.2.3 React

Approach

Once the object was acquired and the more convenient target was found, we have to approach to the object target in order to grasp it. To do this task we have to go close to the target avoiding obstacles like boxes. The position used as target has been acquired in the mapping scene stage and it can be changed due to external factor and position error, so we use it as a raw reference to get closer and then we go to correct the error.

Error correction:

At this point, we are close to the ball and we must have finer precision in order to correctly grip the object. We are going to exploit vision task in order to use closed loop or other techniques which allow us to reduce the error in terms of distance between gripper and object.

To reach this task we have to obtain the distance between gripper and ball acquiring their relative position from depth camera's point cloud. As you can see in figure 8 the center of the red ball represents its corresponding value **Pe** of point cloud (**O** is the center coordinates system of Real Sense). The same for the desired pose **Pd** which was acquired only once, because it remains constant (without considering accidental shift of the camera). We can compute the distance between this two points and adds it as increment for the actual end-effector pose.

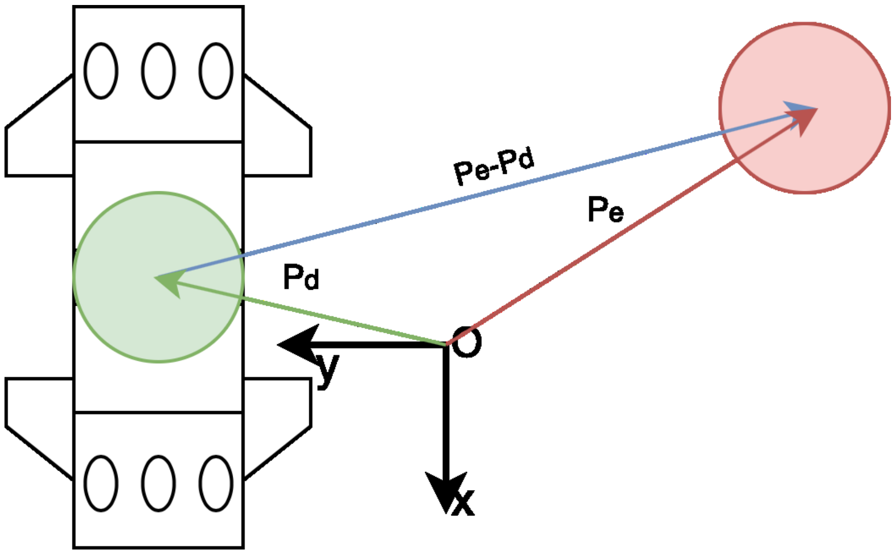
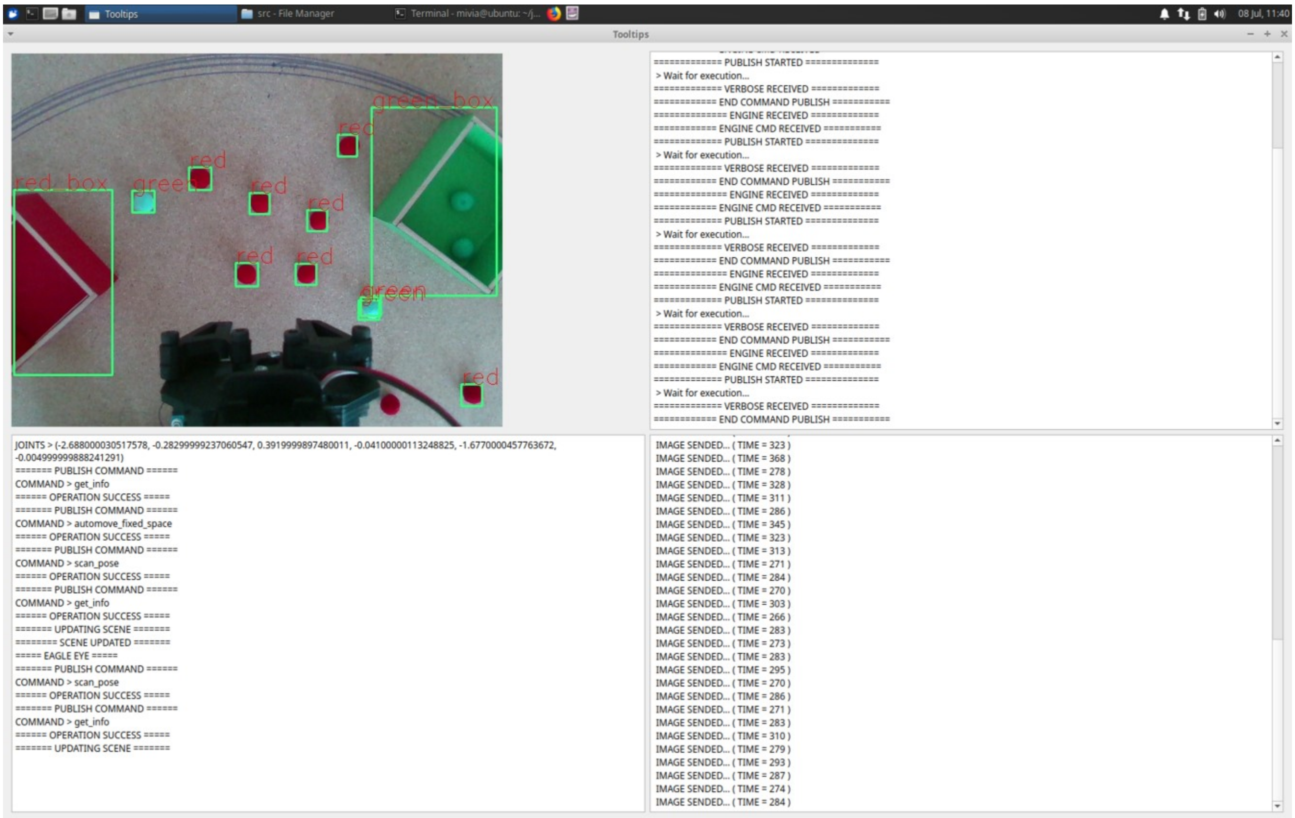


Figure 9: Showing error correction

4.3 Debug

The **debbuger_node** was created to obtain a feedback from each node. We take advantage of the **/rosout** topic, an already existing topic where is published all the logs. We create a graphical interface using PyQt5 library, in which there is one box for each node that publishes information and there is also the image with the bounded boxes.



5. Evaluations

5.1 Network evaluation

The test was performed on a separate test set consisting of 123 images and the obtained results are shown in the following tables:

PREDICTED

EXPECTED

	Red Ball	Blue Ball	Green Ball	White Ball	Red Box	Blue Box	Green Box	White Box	Not Detected
Red Ball	114	4	0	0	0	0	0	0	44
Blue Ball	0	172	0	0	0	0	0	0	40
Green Ball	0	68	31	0	0	0	0	0	12
White Ball	1	9	0	114	0	0	0	0	51
Red Box	0	0	0	0	37	0	0	0	22
Blue Box	0	0	0	0	0	45	0	0	10
Green Box	0	0	0	0	0	2	65	0	10
White Box	0	0	0	0	0	0	0	23	33
Extra Detected	1	18	9	10	0	0	2	2	0

	Red Ball	Blue Ball	Green Ball	White Ball	Red Box	Blue Box	Green Box	White Box	Average Overall Classes
Precision	**0.98**	**0.63**	**0.78**	**0.92**	**1.00**	**0.96**	**0.97**	**0.92**	**0.895**
Recall	**0.70**	**0.81**	**0.28**	**0.65**	**0.63**	**0.82**	**0.84**	**0.41**	**0.6425**
F1 Score	**0.82**	**0.71**	**0.41**	**0.76**	**0.77**	**0.88**	**0.90**	**0.57**	**0.7275**

As you can see from the previous tables, there are many cases where objects are not detected. This happens especially when only a very small portion of the object is present in the image, or when the object is almost completely occluded by another object.

Another problem that can be noticed is that most of the green balls are detected as blue balls, this results in an error during the execution of the task, as the robot will deposit the ball in the wrong container.

This problem is caused by the fact that, contrary to what happens for the boxes, in which the 4 colours are all well distinguishable, the blue balls and the green balls inside the images, captured by the Real Sense camera, appear of a very similar colour, almost indistinguishable even for the human eye, this is caused by the variation of the lighting conditions in which it can be found.

To cope with this problem, it was therefore decided to use a different colour space than the classic RGB to perform the detection and classification tasks.

In particular, the HLS, HSV, YCrCb, LUV, LAB, YUV colour spaces were examined, but what has been found is that the best discriminations between the different colours of the balls (especially with regard to the blue balls and the green balls) are obtained in the RGB colour space.

Furthermore, it was verified whether the removal of one or more channels from the RGB images led to improvements. Therefore, the RG, RB, GB, R, G, B colour spaces were examined, but also in all these cases the different colours were less discriminable than the RGB colour space.

Finally, it was verified if improvements were made with grayscale images, but even in this case the outcome was negative. Therefore, it was decided to continue using the detector trained on the RGB images for the execution of the detection and classification tasks.

Figures