POLITECNICO DI TORINO

Master of Science in Mechatronics Engineering

Politecnico di Torino

Master Degree Thesis

Object recognition and grasping with a UR robot

Supervisors

Prof. Stefano MAURO

Eng. Laura SALAMINA

Candidate

Giovanni Antonio COSSU

Academic year 2021-2022
Torino

Abstract

Visual servoing systems for the control of robotic manipulators are becoming a most relevant issue for robot application in unstructured environments. One of these operations is the capacity of a robotic arm to find the pose of an object and to use that information to pick it up. The goal of this thesis is the recognition of an object by an RGB-D camera image, and the use of a robotic arm to grasp it. To perceive that objective a Realsense D-435 camera and a UR5 robot were used. The RGB image coming from the camera was given as input for the Convolutional Neural Network (Mask R-CNN) utilized for the object recognition task, while the depth image together with a simple inverse projection transformation were used in order to find the X-Y-Z coordinates of the previously found object. This kind of approach was applied because it does not require a hardware with a huge computational power, can have a Network trained on different sets of objects, and it is possible to exploit it on camera models different from the Realsense D-435. An already trained Network was used. At first a simulation environment, called GazeboSim, was considered for preliminary test, and then real world experiments have been done.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Computer Vision

Humans take information about their surroundings with the help of their senses. The sense of sight, it is no secret, is the most studied sense among all the other senses [1].
Because vision is so important, attempts have been made since the 1970s to duplicate human vision electronically using cameras or other types of sensors. In the beginning, it was believed that the “vision task”: "would be only a small problem and it could be solved in just one summer" (1966 Marvin Minisky). Unfortunately, they were completely wrong. The aim of computer vision is to characterise the world in one or more images and to recreate its features, such as form, lighting, and colour distributions. The main point, however, may not be the mere reproduction of vision but, the duplication of all the background work our brain does to process and understand the information coming from our eyes. For a human, observing and the analysing a scene seems effortless, but for a machine it is no easy task at all.

1.1.1 Difficulty in Computer Vision

The main difficulties of computer vision could be summarized in five main points [2]:

1. Loss of information in 3D->2D: this happens normally in cameras. They typically project the real object into an image plane. This projection is done by representing every point of the real object along rays on the

2D plane. This transformation does not preserve angles and collinearity, and due to this, the camera does not distinguish small close object from big far one.

2. Interpretation on images: humans, because of their past experiences and ages of evolution can interpret and distinguish every scenario without any effort. Instead, computers' capacity to understand images is quite bounded, even if Artificial Intelligence (AI) has done tremendous steps toward that direction.

3. Noise: as with every other measurement, also in vision, noise cannot be e underestimated.

4. Too much data: video and Images occupy a lot of memory. Due to this some applications can not be performed in real-time so easily.

5. Local view and global view: understanding the whole context may be difficult since cameras can only catch a tiny portion of the real surroundings.

1.1.2 Hierarchical Organization

Image understanding can be described as the effort to build a connection between input images and previously built models of the environment. This procedure is sorted into a few stages that are divided into a hierarchical organization like the following [2]:

1. Perception: process that provide a computer image

2. Pre-processing: noise reduction and detail information

3. Segmentation: divides the image in object of interest

4. Description: compute characteristic that are useful to differentiate one object from another

5. Recognition: the process that identify an object

Perception

At first, light passes through the camera's optics then it interacts with an imaging sensor. There exist a wide range of sensors, but normally, they all behave in the same manner, capturing the photons and then converting them into a continuous function f(x,y) of two coordinates in the plane. Since a computer cannot deal with continuous function an appropriate data structure, in this case, a matrix must be used to represent an image for computer processing. Therefore the image is digitalized at first sampling f(x,y) and then assigning to each continuous sample a quantized value. It is obvious that there is a connection between the level of detail in the image and the density of the digital sample. Just to have some data a 4k TV has 3840 x 2160 pixels.

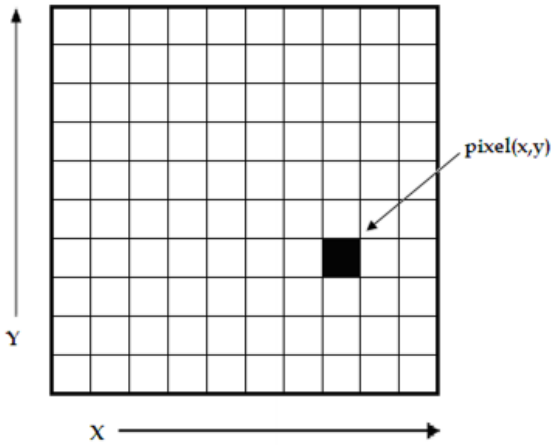

Figure 1.1: Matrix representation of pixels

Pre-processing

At this point, images are still not suitable for further analysis. They still have all the information captured by the sensor. What they need now is to go through the real first step of Computer Vision which is called Pre-processing. Nevertheless, pre-processing is quite helpful in several scenarios because it could be used to suppress all the unrelated data for our process. Example of this process include:

- Filtering

- Color balancing
- Sharpness increase
- Reduction of noise
- Geometric transformation (ex. Rotation of the image)

Concerning Filtering there exist lots of different techniques which deal with the kind of image needed. Neighbourhood average filtering, for instance, given an image f(x,y) creates a filtered picture g(x,y) where the intensity of each pixel is obtained through the average of the intensities of pixels of f in a predetermined neighbourhood of (x,y). One more example of filters is the stencils. They are small bidimensional sets, whose coefficients are selected to reveal a given property or feature in an image (like highlighting isolated points).

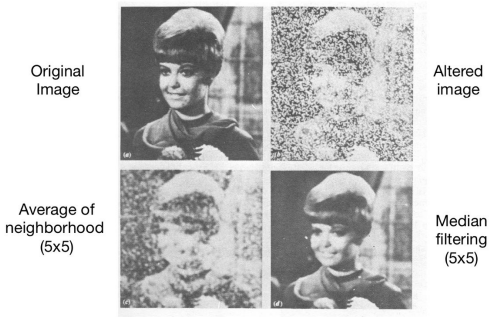

Figure 1.2: Different kind of filtering

Just to have a better view of Pre-processing an interesting procedure to describe is Edge Detection. Using Edge Detection, as it is possible to guess by the name, the main purpose is to find the Edges of objects in a picture. This method is still a processing algorithm and therefore neither the edges nor the things surrounding them are given a logical meaning.

Segmentation

This is the third step of Computer Vision, and it is one of the most important. It allows subdividing the scene into its constituent parts, or objects. Segmentation of an image could be complete, in which a series of images region matching only with the object provided in the input picture is obtained, or partial, in which regions do not exactly match uniquely with the object. Although there is an immediate benefit in a significant reduction in data volume, completely accurate and comprehensive segmentation of complex scenes is typically not possible at this processing step. Therefore, normally, segmentation is partial and then its output is provided to a higher level process which can completely achieve our division aim.
Segmentation may be done using:

- Discontinuity: edge detection
- Similarity: thresholding and region growing

Edge detection segmentation is one of the first techniques used but it is still reliable. Using this kind of segmentation, it is possible to find discontinuity in grey-scale level, colour, or texture. This process, though, is not enough to be considered a proper segmentation. This process must be followed by an edge combination where all the edges are connected into an edge chain. The goal is to segment an image at least partially, which is accomplished by grouping local edges into an image where only edge chains that match existing objects or image sections are present. Although this process, as said before, is quite reliable, it has a series of problems due to image noise or unsuitable information in an image. Concerning thresholding, the grey-scale level one is the simplest segmentation process. When looking at a picture, humans can notice that some parts of it are characterized by constant reflectivity or light absorption. Starting from that idea it is not difficult to find a brightness constant or threshold to properly divide objects in the scene and background. Since it is a fast process, it can be performed in real time and due to this,

even though it is simple, it is still widely used. In the following lines, a simple Algorithm is reported for thresholding just to make have an idea of how it works.

Algorithm 1 Basic thresholding

1: **procedure** THRESHOLDING
2: ▷ In the picture f, look for all of the pixels $f(i, j)$. If $f(i, j) \geq T$, a segmented image pixel $g(i, j)$ is an object pixel; otherwise, it is a background pixel.
3: **end procedure**

A different segmentation technique is called Region-oriented. The segmentation approaches described above are based on discontinuity. Looking for discontinuities means seeking boundaries between regions, instead in this case the whole region is what the technique looks for. The first procedure which falls under region-oriented segmentation is called Region growing. Starting with a collection of seed points, areas are expanded by including nearby pixels with comparable properties (e.g., intensity, colour, texture, etc.). Two main problems can be immediately spot:

1. Seed selection
2. Choice of the characteristic to be used for aggregation

Description

The extraction of an object's characteristics is the description challenge in the visual process. After that, these traits will be helpful for object recognition. One thing to take care of is that the description problem should be independent of the object's dimension, position, and orientation. But it is not the only to take care of, it also should have enough information to distinguish unequivocally one object from another.

Recognition

Recognition is the last step of the Computer Vision task. To perform recognition, one more ingredient is needed, which is some knowledge about the object to classify. Without that information, it is impossible for any kind of algorithm to recognize an object.

This step recognizes some kind of pattern or similarity between the previous knowledge about an object and the result of all the other steps above. Pattern recognition can be divided into two different techniques which are:

- Supervised: examples on the correspondence between pattern and target can be provided
- Unsupervised: only a pattern is provided and the chosen algorithm divides the present pattern in classes

A different way to distinguish the different pattern recognition techniques could also be a division done depending on what drives the algorithm, for example:

- Rule-driven: In this case a rule is assigned for each kind of object (e.g., and object have this characteristic...) For this there are different ways to assign rules:
  - Deterministic
  - Probabilistic classifier (e.g. Bayesian)
  - Fuzzy logic system
  - ...
- Data-driven: one example to this could be a Neural Network (deep-Neural Network,Convolutional Neural Network,etc.)
- Hybrid: it is a mixed version of the techniques listed above (neuro-fuzzy,fuzzy-clustering)

In order to perform the recognition of an object at first the algorithm must be trained with some data that possibly are representative of all possible situations and then a Test and Validation phases are required. Also for these two last phases, two different distinct datasets are necessary.

1.2 State of the Art

The main problem with robot grasping is that there are sources of uncertainty that do not allow the robot to perform a perfect grasp [3]. Those are:

- Perception uncertainty: cameras give a continuous stream of images, but most of the time the information arriving at the robot is not noiseless. Therefore the perfect pose estimation of an object from a camera is not always possible.
- Physics uncertainty: objects have friction, masses, etc. Therefore sometimes those factors influence objects grasping
- Control uncertainty: robot actuators do not perfectly move as set.

Therefore taking into consideration these factors a lot of solutions have been exploited, during the years.

1.2.1 *2-D*

In the late 90s/early 2000s depth camera were really expensive. Considering this research focused on 2D images for object recognition. At that time, the main algorithms for object recognition were those based on feature descriptors. Pose estimation was possible thanks to some previously saved models of the recognized object. This means that the object pose could be found only if a model of the object was available and it could be compared with the previously found object.
During the same period, there was the development of some new computer vision algorithm which can be used for tasks such as:

- Object recognition
- Image registration
- Classification
- 3-D reconstruction

These Computer Vision algorithms are able to take images as input and, after some processes, output feature descriptors or feature vectors. The idea behind them is to obtain a series of features which are invariant to image transformation, so these information can be found even if the image is distorted.

Since, over time, a lot of algorithms were created, it is worth list some of them just to have a brief overview of what has been developed:

- Scale-invariant feature transform (SIFT) [4]
- Speeded Up Robust Features (SURF)[5]
- Binary Robust Independent Elementary Features (BRIEF) [6]

The ones listed above are just a small group of those which were developed during that time. In figure 1.3 there is an example of a feature extractor using SIFT.

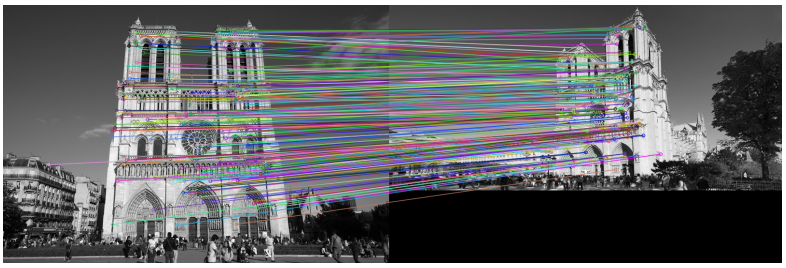

Figure 1.3: Feature extraction using SIFT

It has found a series of features, which even if the picture on the right was taken from a different perspective, can be found and used for the recognition of the wanted object.
When common features have been found, therefore recognition has been performed, it is possible to estimate the pose of the found object. To do so a data set, where all the models' reference frames are stored, is essential. Once one object is identified the rotation matrix and translation vector, which define the relationship between the two coordinate frames, can be calculated using these datasets. Unfortunately, the uncertainty caused by improper point matching and noisy points makes it impossible to estimate an object's posture with enough accuracy.

1.2.2 *3-D*

Point Cloud

After the early 2010s and the release of cheaper depth cameras changed everything. Figure 1.4 is one example of a depth camera. The depth information could be used for pose estimation of the object (Information coming from a depth camera can be seen in figure 1.5). These information coming from the camera can be converted into a Point Cloud, which is a set of points in space. Each of these points has its X-Y-Z position. An example of a Point Cloud is shown in figure 1.6.
From this concept, a lot of algorithms using point cloud came out.

Figure 1.4: Microsoft Kinect

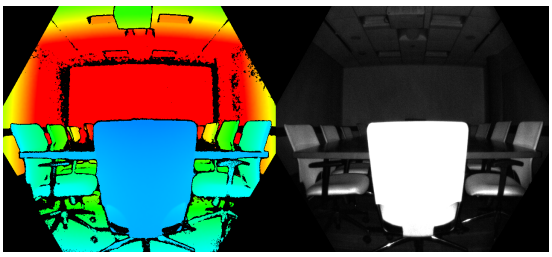

Figure 1.5: Depth Image

The majority of 3-D object recognition algorithms which uses point cloud works using a model-based approach. 3-D representations of the potential object of interest are first processed offline. This first process creates a data set of extracted features that in future can be used for object recognition. Therefore features are extracted from a point cloud and then given as input to a recognition algorithm. These features are then compared to the pre-built data set to perform object recognition. A lot of feature extractors were developed:

- Signature of histograms of orientations (SHOT) [7]
- Rotational projection statistics (RoPS) [8]
- 3-D tensor [9]

Once the object is identified, it is possible to do pose estimation. Also in this case, an external data set can be used to calculate the rotation and translation matrix.

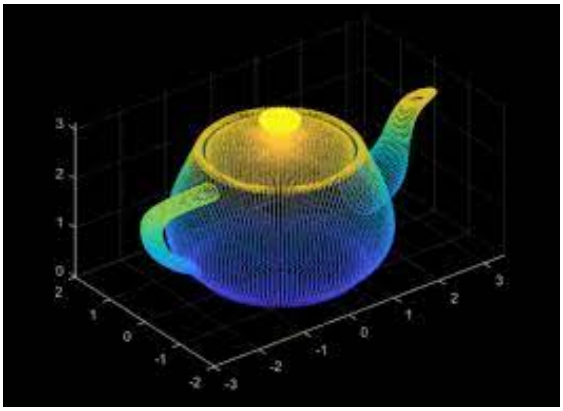

Figure 1.6: Example of a Point Cloud

Machine Learning Approach

After 2012 when A.Krizhevsky implemented Deep Convolutional Neural Network (CNN) within a Graphics Process Unit (GPU) [10], image recognition and pose estimation grew exponentially. From that time on, a lot of Neural Networks (NN) came out. In this case, there are three main approaches. One directly calculates the point from which the object can be picked up without the direct recognition of the object in the scene; the second recognizes the object and calculates its pose without any proper grasping point calculation; the last is a mix of the previous two. There are a lot of examples of networks which try to perform the wanted task. Object recognition, as stated, in the previous chapter, is not a new task, but for the first time, this problem could be solved efficiently. This approach was a revolution, it opened a new way of object recognition, but also for object grasping. In the following years, more and more evolved Convolutional Neural Networks came out. Some of those networks were not only able to find and recognize a specific object but were also capable of evaluating the best point for grasping an object.

Chapter 2

Tested Algorithms

Throughout the years a lot of algorithms for object recognition and pose estimation were developed. Some of them have been tested. The algorithms need to work with a small computational power, a limited memory storage and at the same time have to be fast enough.

2.1 Relasense D435

It is necessary to briefly introduce the camera used for image recognition and pose estimation because it is the main piece of hardware to perceive the aim of this work. An **Intel® RealSense™ D435** was used [11]. Figure 2.1 shows the aesthetic while figure 2.3 shows the camera dimensions. This camera was chosen because of its depth module. The Intel® RealSense™ depth camera D435 provides high-quality depth images for a range of uses. Applications like robots or augmented and virtual reality, where seeing as much of the scene as possible is crucial, its wide field of view is ideal. Since the camera has a compact form factor it can easily be used in a big variety of scenarios.

Figure 2.1: Intel® RealSense™ depth camera D435

To provide depth image, it uses stereo vision. A left imager, a right imager, and an optional infrared projector make up the stereo vision implementation. In low-texture environments, the infrared projector creates a non-visible static infra red pattern to increase depth perception. The left and right imagers take pictures of the scene and send the information to a depth imaging (vision) processor, which uses the information to calculate the depth of each pixel in the image by comparing points on the left and right images and by shifting between points on the left and right images. This process of depth estimation is perfectly summarized in figure 2.2

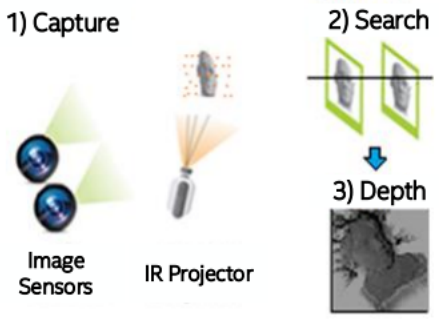

Figure 2.2: Depth implementation

For the aim of this work, it is essential to know the minimum distance from camera to object for which it provides reliable depth data. In table 2.1 the minimum distances, depending on the camera resolution, are reported.

Table 2.1: Minimum Depth Table

Resolution	D400/D410/D415 Min-Z [mm]	D420/D430 Min-Z [mm]
1280x720	450	280
848x480	310	195
640x480	310	175
640x360	240	150
480x270	180	120
424x240	160	105

The resolution was set to $1280x720$, therefore the minimum distance is 280mm. If the objects distance is less than 280mm, it is not possible to rely on the output of the camera.

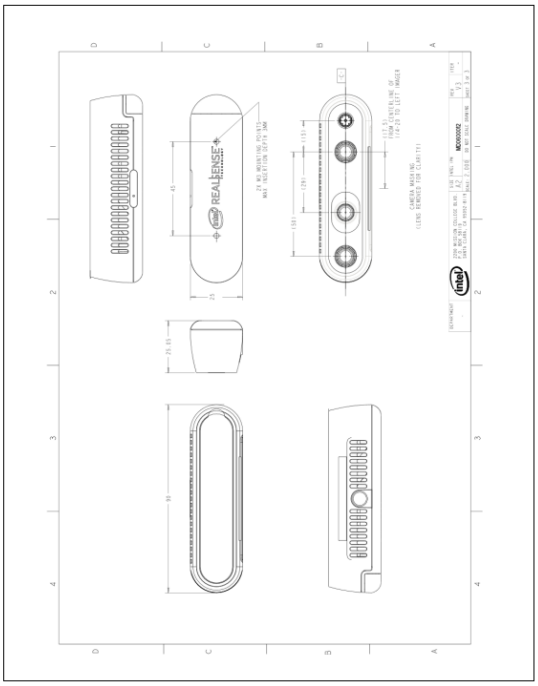

Figure 2.3: Intel® RealSense™ Depth Camera D435

2.1.1 Realsense Pipeline code

The camera was used via a pipeline [1] [12] developed with a python library called `pyrealsense2`. A `RealsenseCamera` class was defined to have all the necessary configurations enabled at the same time.
The next part of the reported code allows starting both the depth sensor and the RGB module. Once the images are processed they are converted to NumPy [2] arrays as required by the used Neural Network.

```
class RealsenseCamera:
    class RealsenseCamera:
    def __init__(self):
        # Configure depth and color streams
        print("Loading Intel Realsense Camera")
        self.pipeline = rs.pipeline()


        configuration = rs.config()
        configuration.enable_stream(rs.stream.color, 1280, 720,
  rs.format.bgr8, 30)
        configuration.enable_stream(rs.stream.depth, 1280, 720,
  rs.format.z16, 30)

        # Start streaming
        profile=self.pipeline.start(configuration)
        aligned_color = rs.stream.color
        self.align = rs.align(aligned_color)
        self.intr=profile.get_stream(rs.stream.color).
  as_video_stream_profile().get_intrinsics()
```

The above code enables the camera streaming with a resolution of 1280x720, for both the color and the depth streaming.

```
        depth_image = np.asanyarray(filled_depth.get_data())
        color_image = np.asanyarray(color_frames.get_data())
```

[1] A pipeline in software engineering is a series of processing components (processes, threads, coroutines, functions, etc.) structured so that each element's output serves as the subsequent element's input; In this case the pipeline simplifies the user interaction with the device and computer vision processing modules

[2] NumPy is an open source project aiming to enable numerical computing with Python.

Using `np.assanyarray` the colour and depth frames are converted to NumPy arrays. This conversion, as explained before, is the input to provide to the NN model to make object recognition possible.

2.2 Algorithms

2.2.1 Find Object

Most of the algorithms used in 2D object recognition can be found in an open-source computer vision library, called **OpenCV**, and can be tried using a simple interface called **Find-Object**. In the search for a good implementation to estimate the pose of an object, a ROS (section 3.1) integration package of the **Find-Object** application was tried. This package is quite simple to use. In a first phase, it displays the camera streaming and applies this scene to the most important features of the framed objects. From the same window, it is possible to cut out a certain area that must contain the wanted object that later on has to be recognized. In this way the algorithm should ideally find the object when it appears in another video/image. If a camera with a depth sensor is available, it can also estimate the pose of the detected object. Theoretically, this approach seemed to be suitable for the purpose of this work, but by doing some measurements some issues were encountered. If the object does not have a lot of particular features, the package is not able to distinguish it from the rest of the scene; so the pose estimation, due to the previous problem, is not precise. And, even if the object is found, the pose estimate is completely wrong if it is at the edge of the image. Since this package was tested with the Realsense D-435, some preliminary measurements were made to evaluate the behaviour from different distances. These measurements were made on a table covered with a graph, to keep track of the position of the object and with the background made of cardboard. The background was made of cardboard to ensure that no additional features could interfere with the algorithm's pose estimation. During the experiment, the camera was held in one place and the object was moved to different predetermined positions. By measuring the distance of the object from the camera the actual X-Y-Z coordinates were quite reliable. These distances were then compared to the distances by the algorithm.
The measurements were saved in an Excel, shown table 2.2

Table 2.2: Find Object Measurements Table

Position D-435	X	Y	Z	x_mis	y_mis	z_mis
CentralPos 1	37	0	0	36	-2	-7
CentralPos 2	37	17	0	55	36	-11
CentralPos 3	37	10	0	38	13	-7
CentralPos 4	37	5	0	38	7	-7
CentralPos 5	37	3	0	37	4	-7
CentralPos 6	37	-12	0	54	-24	-9
CentralPos 7	37	-7	0	53	-12	-9
CentralPos 8	37	-4	0	37	-4	-6
CentralPos 9	31	0	0	28	0,8	-6
CentralPos 10	31	17	0	0	0	0
CentralPos 11	31	10	0	31	15	-7
CentralPos 12	31	5	0	30	8	-6
CentralPos 13	31	3	0	31	5	-6
CentralPos 14	31	-12	0	54	-28	10
CentralPos 15	31	-7	0	54	-17	-11
CentralPos 16	31	-4	0	32	-5	-6
CentralPos 17	46	0	0	46	-1	-6
CentralPos 18	46	17	0	54	34	-9
CentralPos 19	46	10	0	56	23	-9
CentralPos 20	46	5	0	45	11	-7

The above table is a part of all the measurements that were taken. The X-Y-Z coordinates are the actual positions of our object while x-mis y-mis and z-mis are the positions estimated by the algorithm. The results were plotted on Matlab.

The three figures 2.4, 2.5, 2.6 show that none of the calculated position by the algorithm can be considered reliable.

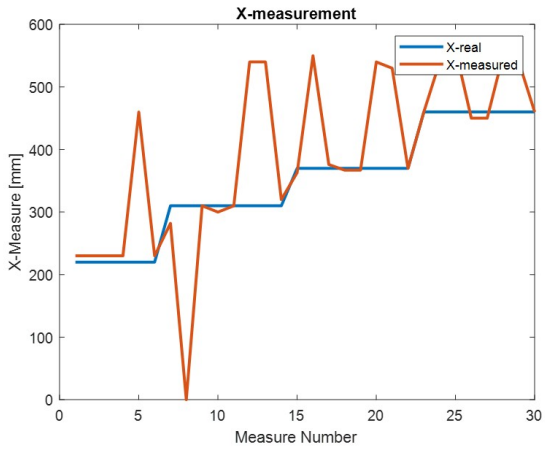

Figure 2.4: Real and Measured Value on X axis

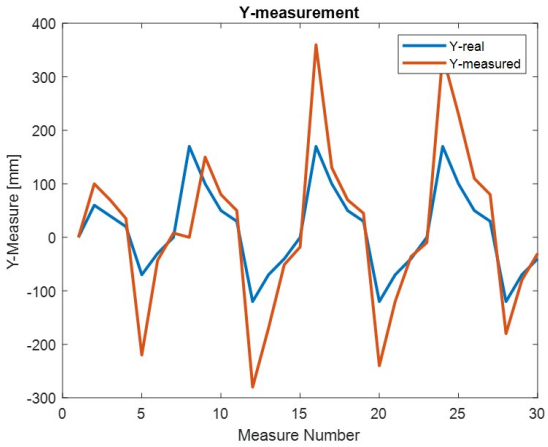

Figure 2.5: Real and Measured Value on Y axis

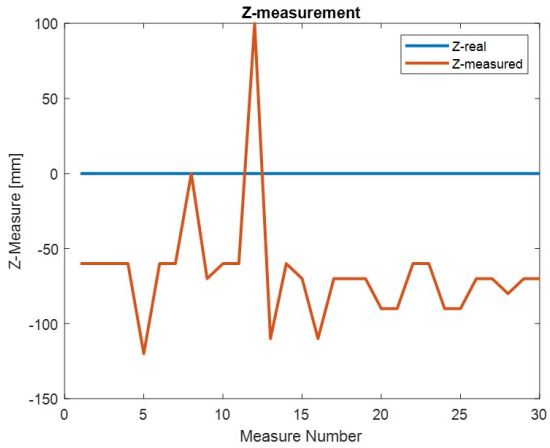

Figure 2.6: Real and Measured Value on Z axis

In some measurements the estimated position is really far away from the real one, normally this happens when the object is near the edge of the camera field of view. Therefore, a big drift in the 3D pose estimation is encountered.

Figure 2.7 shows the percentage of the relative error made by the algorithm. In this case, the error is just the difference between the real value and the measured one, normalized to the real value.

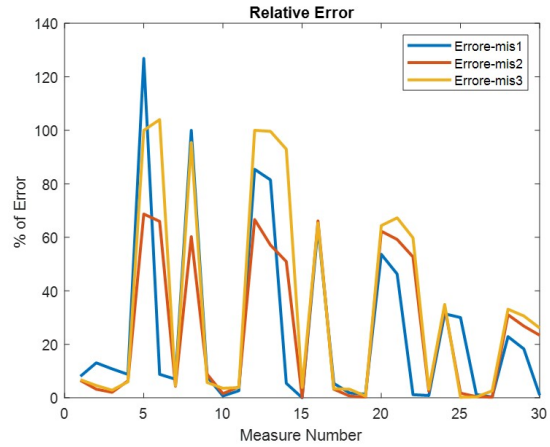

Figure 2.7: Relative Error between Real and Measured value

The Relative Error graph shows that in some places, where the object was

moved with respect to the camera, the error goes up to more or less 130%. Therefore, after these experiments, this algorithm was discarded.

2.2.2 Point Cloud Library

The Point Cloud Library (PCL) is a standalone, large-scale, open project for 2D/3D image and point cloud processing [13]. The Point Cloud Library collects all the algorithms which concern Point Cloud processing and stores them all in one place [14]. A specific PCL object recognition and pose estimation pipeline was tested. It uses the *pcl_recognition* module, and it aims at performing 3-D Object Recognition, also providing a transformation matrix identifying the 6DOF pose of the recognized model as output. This kind of recognition, after 3D description matching [15], where descriptors between the current scene and the model in the library are computed, uses Correspondence Grouping algorithms that group correspondences that are geometrically consistent in the clusters and discard the ones that are not. Unfortunately, during testing, a lot of issues were encountered.

Most of the time, since the algorithm needed a model of the object to be found, it was not able to find any kind of correspondence between the model that was given and the current scene. This mainly depended on two different factors:

- The Point cloud model taken from the camera was noisy;
- During the implementation of the algorithm, it is possible to tune some parameters to have better object recognition and pose estimation. The main problem is that those parameters differ for every given input, and the only way to tune the is with a trial and error process.

These issues, together with the fact that it was not suitable to gather a point cloud model for each item, led to the conclusion that this technique was unreliable and unusable. Therefore, also this algorithm was discarded.

2.2.3 Neural Networks

Different kinds of Neural Networks have been investigated. All the Networks were designed either to identify the object, and the pose could be estimated using the camera or they could directly find a good grasp point.

Grasp Pose Detection

The **Grasp Pose Detection** [16] selects a grasp candidate from a point cloud, and applies a CNN to determine if the selected candidate would be successful. Unfortunately, even if it seemed a good candidate when the code was implemented a grasp candidate was not possible to be found for each point cloud file that was tested.

Dex-Net

The second implemented Neural Network was the Grasp Quality Convolutional Neural Networks (GQ-CNN) [17]. This NN is part of a project called Dex-Net whose goal is to develop highly reliable robots grasping across a wide variety of rigid objects. Figure 2.8 shows how the Dex-Net Architecture works. This network uses a dataset of 6.7 million synthetic point clouds, grasps, and associated robust grasp metrics, computed with DexNet 1.0. The resoults are used to do offline training to the Grasp Quality Convolutional Neural Network (GQ-CNN). GQ-CNN predicts the robustness of the candidate grasps from depth pictures. Pairs of antipodal points in a 3D point cloud created by a depth camera identifies a collection of several hundred grasping possibilities. The GQ-CNN quickly identifies which grip candidate is the strongest. In this case, since a pre-trained Network was necessary to run the CNN, and neither one was found nor was possible to train a new one (since a powerful GPU is necessary) it was not possible to use it. Therefore, in this case no tests were performed.

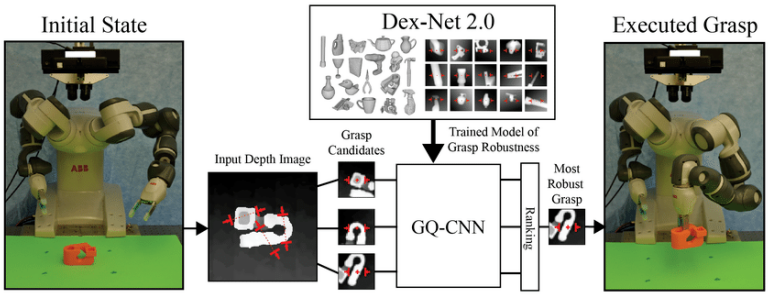

Figure 2.8: Dex-Net Architecture

Antipodal Robotic Grasping

The third CNN implemented, tackles the problem of generating robotic grasping for an object is the Generative Convolutional Neural Network (GR-ConvNet) [18].

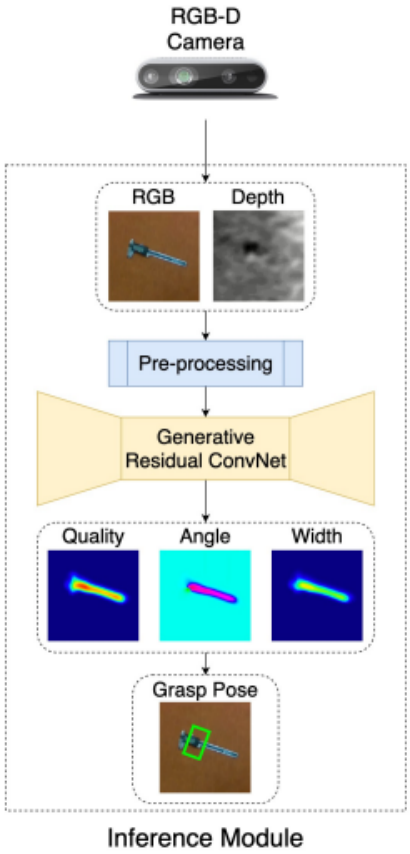

Figure 2.9: Antipodal Grasping Architecture

As explained in figure 2.9, an RGB and aligned depth image are acquired from an RGB-D camera. At that point, the image is pre-processed to match the input of the Neural Network. The Generative Residual ConvNet has as outputs quality, angle and width images, which are then used to infer the grasp pose. Using this information together a grasp pose for the object can be found. Concerning the code implementation of this NN, the installation was performed and all the requirements were fulfilled. Although the grasping point seemed good, it was decided not to use this approach.

Deep Object Pose Estimation

Going on with Convolutional Neural Network also the DOPE ROS package was considered [19] for object recognition and 6-DoF pose estimation from an RGB camera. In this case, since an Invidia GPU was necessary, it was not even possible to fulfil all the requirements. Considering that, it can be considered as an alternative if a GPU is available.

Mask-RCNN

Since a simple and fast approach was required, Mask-RCNN [20] was tried. An implementation of this network was tested and, as it is going to be explained later, this was the chosen approach to perceive the aim of the work. The Mask-RCNN is able to efficiently find and identify an object in a shown environment. The identification is not the only output: this Network can also draw both a bounding box [3] and a mask around the object.
The 3D pose estimation was done by combining the bounding box information and the depth measurements coming from the camera. So the XYZ coordinates of the desired object were estimated.

[3]A bounding box is the smallest box which is possible to draw around an object to completely delimit it

2.3 Introduction to Region-Convolutional Neural Network

The Mask-RCNN was the selected Neural Network for object recognition. An already trained network was used. The dataset on which the network was trained is the COCO dataset `mask_rcnn_inception_v2_coco_2018_01_28.pbtxt` [21]. Considering this, only a specific class of object can be recognized. Among all of these **sports ball** and **bottle** were selected to perform the experiments. Before going into details with the used Mask-RCNN and the showing results of the performed measurement, a brief introduction to the evolution of the R-CNN network is reported. As already said, recognition is the last step of the *Computer Vision task* and the use of Neural Network is one of the possible alternatives to perform it.

2.3.1 Evolution of R-CNN

The use of CNN was really popular in the 1990s, but then dropped making space for support vector machine. However, after 2012, when Krizhevsky [10] was able to use GPU to train and evaluate Networks, the interest in NN blew up again. In 2014 the first R-CNN came out [22]. It is called R-CNN because it combines region proposal with CNN. This new approach can be divided into three modules:

1. The first creates a category-independent region proposal. [4]

2. The second is a large convolutional neural network that extracts a fixed length feature vector from each of the previous regions.

3. The third is a series of Support Vector Machines (SVMs)[5] which identifies the object's class.

Figure 2.10 perfectly describes the R-CNN process.

[4]The region proposal process identifies a section of the image where there is the possibility to find an object

[5]SVMs are supervised learning models that use learning algorithms to examine data for classification and regression

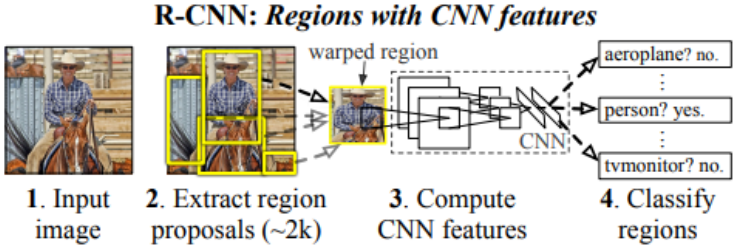

Figure 2.10: R-CNN regions

Doing so, from an input image, it is possible to find and categorize a series of objects. Of course object recognition, as stated in previous chapter, is not a new task, but for the first time, this problem could be solved efficiently and robustly. In the following years, an evolution of that network came out. It was called **Fast R-CNN**, and introduced several innovations which improved both training and testing speed while increasing also detection accuracy. The architecture of **Fast R-CNN** [23] is quite different from its ancestor **R-CNN**. As shown in figure 2.11, the first stage of **region proposal** is done using several convolutional and max pulling layers, which have as output a convolutional feature map. The convolutional layers have a series of filters which can identify specific patterns inside the proposed region, whereas the max layers, depending on the size of the used filter, can reduce the input dimension. Then, from the feature map, a features vector is extracted. Each of these vectors is fed into a *Fully connected network* which outputs the classes of the found objects and the bounding boxes around the objects themselves.

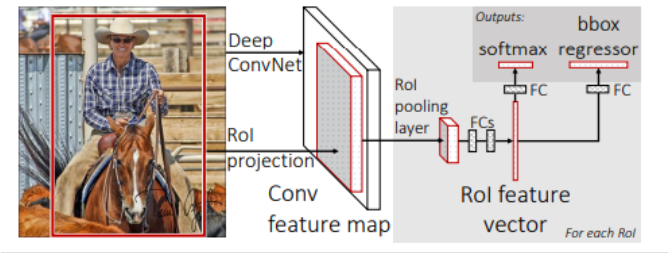

Figure 2.11: Fast R-CNN architecture

In 2016, an improved fast R-CNN came out. It was called Faster R-CNN [24]. This new network is composed of two modules. Figure 2.12 shows that the input image passes through a fully connected CNN. This network creates a feature map, which is given as input to the Region Proposal Network (RPN), the first module. The RPN creates, as output, a series of regions, which should be, the sections where the object is located. RPN also gives an object score, calculating the probability that the object is present in that section or not. The RPN output is given as input to a fast R-CNN, the second module. The peculiarity of this new approach is that, although there are two sections, some layers of the fast R-CNN and the RPN are shared. Since these two modules have some parts that are shared and some that are not, to improve the training phase, it is possible to train once the shared layers, not changing the layers that belong only to the RPN, and then train the layers of the RPN, keeping fixed the shared layers.
The novelty of this network is that the regions where the Fast R-CNN has to search are directed by the RPN module.

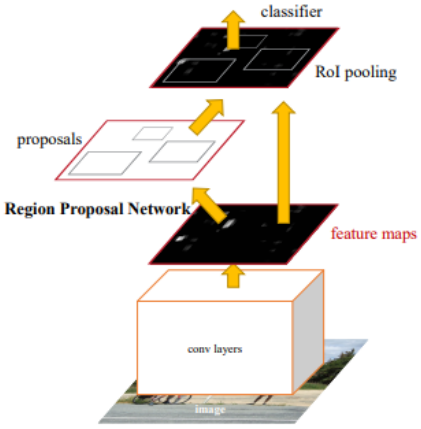

Figure 2.12: Faster R-CNN architecture

2.4 Mask R-CNN

The Mask R-CNN [20] came out in 2018 and extended the faster R-CNN by adding a branch. This branch is composed of some separated layers specialized in predicting the objects' mask. The previously described faster R-CNN for classification and bounding box calculation are still present. In figure 2.13 the distinction between the newly added branch and the faster R-CNN section is shown. From the input image, the Region of Interests in which an object might be located are first selected, then these regions passes through some convolutional layers. At this point, the results of the previous calculation are split into two different branches. One is able to estimate the object classes and the object bouding boxes, the second calculates the masks around each object found. Therefore, in addition to the faster R-CNN outputs, a new one has been added. From figure 2.14 it more evident the division between the two branches. This figure shows two different implementations of the Faster R-CNN, but in addition two new branches have been added that can calculate the mask around the object.

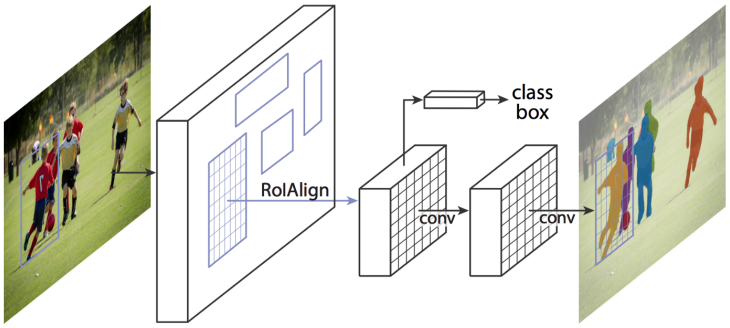

Figure 2.13: Mask R-CNN

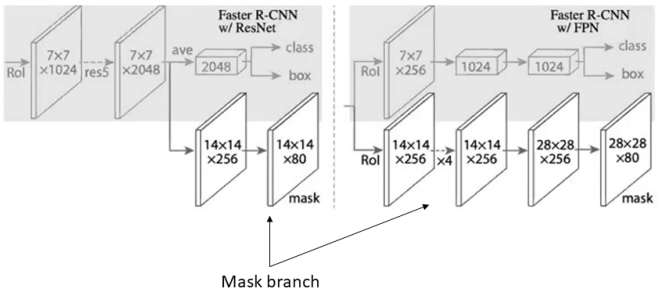

Figure 2.14: Branch Mask R-CNN

2.4.1 Code Overview

Mask R-CNN Code

A Python code [12] was used to implement the Mask R-CNN for object recognition. It is worth taking a closer look at some of the lines of code better understand what has been done here.

```
self.net = cv2.dnn.readNetFromTensorflow("/home/giovanni/
  Desktop/Python_script/dnn/frozen_inference_graph_coco.pb","/
  home/giovanni/Desktop/Python_script/dnn/
  mask_rcnn_inception_v2_coco_2018_01_28.pbtxt")
```

In the beginning, some settings and file loading are necessary to make the code work. The first thing to do is to load pre-trained files, to use the network. In this case, since a **TensorFlow** [6] implementation is used, it is necessary to load a .pbtxt file, which contains the pre-trained weights and, a .pb file, where the model graph is stored.

```
with open("/home/giovanni/Desktop/Python_script/dnn/classes.txt"
    , "r") as file_object:
            for class_name in file_object.readlines():
                class_name = class_name.strip()
                self.classes.append(class_name)
```

For the following passage all the classes, for which the Network is trained, have to be put into a list. All the pre-trained classes are inside a .txt file called **classes.txt**. The elements saved inside this file are the objects that the Neural Network is able to identify.

```
def detect_objects_mask(self, bgr_frame):
        blob = cv2.dnn.blobFromImage(bgr_frame, swapRB=True)
        self.net.setInput(blob)
        boxes, masks = self.net.forward(["detection_out_final",
    "detection_masks"])
        # Detect objects
        height, width, _ = bgr_frame.shape
        detection_counter = boxes.shape[2]
        # Object Boxes
        self.object_boxes = []
        self.object_classes = []
        self.object_centers = []
        self.object_contours = []
        for i in range(detection_counter):
            box = boxes[0, 0, i]
```

[6]Tensorflow is a complete open source machine learning platform. It has a wide range of adaptable tools, libraries, and community resources.

```
15              class_id = box[1]
16              score = box[2]
17              color = self.colors[int(class_id)]
18              if score < self.detection_threshold:
19                  continue
20              # Get box Coordinates
21              x = int(box[3] * width)
22              y = int(box[4] * height)
23              x2 = int(box[5] * width)
24              y2 = int(box[6] * height)
25              self.object_boxes.append([x, y, x2, y2])
26              center_x = (x + x2) // 2
27              center_y = (y + y2) // 2
28              self.object_centers.append((center_x, center_y))
29              # append class
30              self.object_classes.append(class_id)
31              # Contours
32              # Get mask coordinates
33              # Get the mask
34              mask = masks[i, int(class_id)]
35              reg_of_int_height, reg_of_int_width = y2 - y, x2 - x
36              mask = cv2.resize(mask, (reg_of_int_width,
    reg_of_int_height))
37              _, mask = cv2.threshold(mask, self.mask_threshold,
    255, cv2.THRESH_BINARY)
38              contours, _ = cv2.findContours(np.array(mask, np.
    uint8), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
39          self.object_contours.append(contours)
40      return self.object_boxes, self.object_classes, self.
    object_contours, self.object_centers
41          .....
```

After the setting passages, the **RGB frame**, coming from the Realsense camera, can be used to detect the bounding boxes and the masks of the objects. For this purpose, the method `detect_object_mask` is used.
Once the object is found, a bounding box is available to surround it. *(center_x,center_y)* are the pixels coordinates of the centre of the bounding box. The two pixels coordinates will be the two basic reference data for the 3D position estimation of the detected object. Figure 2.15 shows the output of the Mask R-CNN. As it is possible to see, some objects are detected by the Neural Network. For each object, a bounding box is drawn around it.

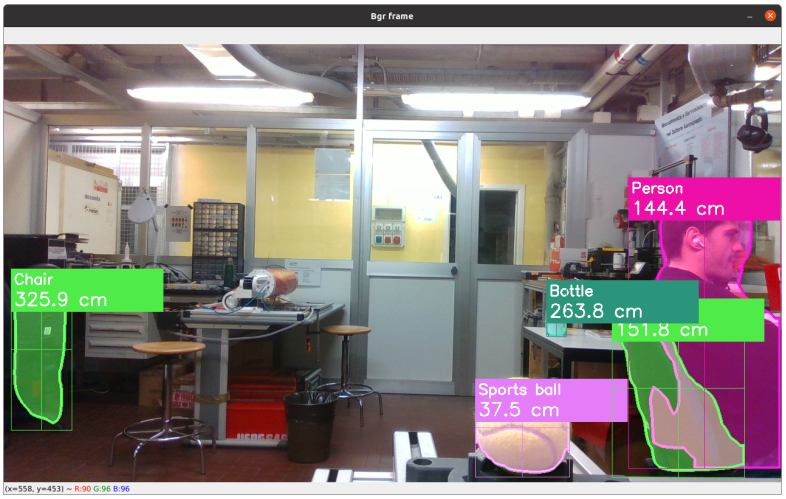

Figure 2.15: Output Image Mask R-CNN

3D Pose Estimation

After the recognition passage, the next step is the estimation of the X-Y-Z coordinates. Up to this point, the Realsense D-435 camera was only used as a regular camera not making use of the depth sensor. The distance to the centre of the bounding box may be calculated using this feature.
Since in the camera coordinate system, as shown in figure 2.16, the depth value of the pixel corresponding to *(center_x, center_y)* is the Z coordinate, one of the three values is found. The other two coordinates, X and Y, are still missing.

Figure 2.16: Camera Reference system

Let's introduce the geometrical model of the camera [25]. As can be seen in figure 2.17 in the image on the left **X**, a 3D point, is projected onto a 2D plane (the image plane). An example of this projection can be visualised by following the line connecting **X** to **x**. Therefore, the projection has transformed **X** into **x**.

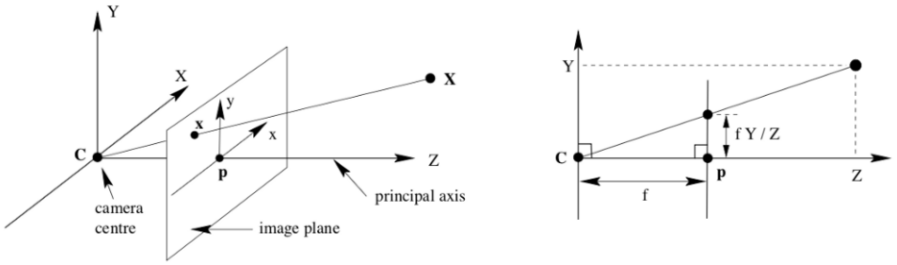

Figure 2.17: Camera Projection Geometry

In order to carry out this mapping of **X** to **x** (or (x,y)=f(X,Y,Z)), some intrinsic parameters of the camera are required.

- Focal length (fx,fy): the distance between the camera centre and the image plane times the pixel density. It describes the angle of view of a

lens. It changes when the pixel resolution of the camera varies.

- Principal point (x0,y0): the centre of the image plane.

Therefore, the formula to perform that mapping is given by:

$$x = \frac{X}{Z} * fx + x0 \tag{2.1}$$

$$y = \frac{Y}{Z} * fy + y0 \tag{2.2}$$

This is how a 3D point is projected onto a 2D plane. In this way, however, the depth information is lost, since each point on the line connecting the (X, Y, Z) with (x,y) is mapped at the same point on the image plane. The line connecting the 2D point to the 3D point is shown in 2.18.

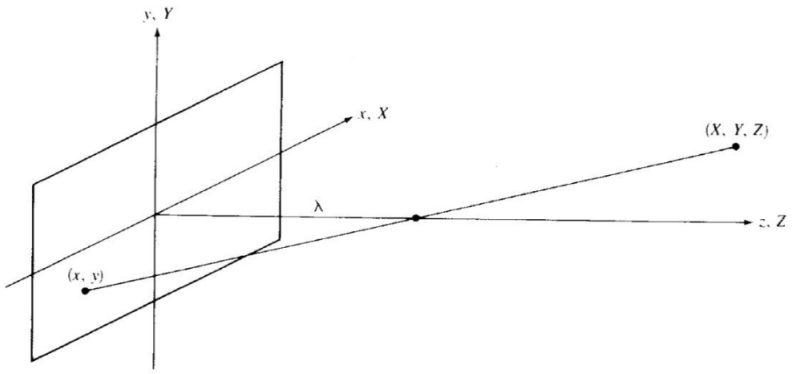

Figure 2.18: From 3D to 2D projection

The reconstruction of a 3D point from a bi-dimensional image requires knowledge of at least one of the coordinates of the 3D point. In this case, the value of the Z coordinate is determined using the depth sensor of the Realsense camera, as mentioned earlier. Therefore, it is possible to find the two missing coordinates by performing an inverse projection transformation on the camera image of the scene. Knowing the centre of the bounding box and having the distance of the object from the camera, that is the Z coordinate, as well as some intrinsic parameters of the image (focal distance and centre of projection), the X and Y coordinates can be calculated using the following equations:

$$X = \frac{depth * (cx - ppx)}{fx} \tag{2.3}$$

$$Y = \frac{depth * (cy - ppy)}{fy} \tag{2.4}$$

In Equations 2.3 and 2.4 the parameters are:

- depth: the distance between the camera and the centre of the bounding box;
- cx: x coordinate of the centre of the bounding box expressed in pixel;
- cy: y coordinate of the centre of the bounding box expressed in pixel;
- ppx: horizontal coordinate of the principal point of the image, as a pixel offset from the left edge;
- ppy: vertical coordinate of the principal point of the image, as a pixel offset from the top edge;
- fx: focal length of the image plane, as a multiple of pixel width;
- fy: focal length of the image plane, as a multiple of pixel height;

Although, all 3 coordinates of the object have been retrieved, one more passage has to be done. The camera might have some inclination or the X, Y and Z coordinates might not start exactly in the middle of the camera. This means that some compensations are necessary.

```
1 Xtemp= dist*(center_x- intrinsic_param.ppx)/intrinsic_param.fx
2 Ytemp=dist*(center_y-intrinsic_param.ppy)/intrinsic_param.fy
3 Ztemp=dist
4
5 self.Xtarget = Xtemp - 17.5
6 self.Ytarget=(Ztemp*math.sin(0)+Ytemp*math.cos(0))
7 self.Ztarget= Ztemp*math.cos(0)+Ytemp*math.sin(0)
```

To compensate for this fact, as it is possible to see in the code , 17.5 mm are subtracted from the estimated X value. The Y and Z coordinates, on the other hand, are adjusted due to the possible camera tilt.

Main Object Recognition and Pose estimation

The code described in the previous section was part of a class called `MaskRCNN` [12].
Methods and objects of this class are then used in the main code [12].

```
ret, bgr_frame, depth_frame, depth_frame_rs = rs.get_frame_stream()
 Intrinsic=rs.intr
 print(Intrinsic)
 # Get object mask
 boxes, classes, contours, centers = mrcnn.detect_objects_mask(bgr_frame)
 # Draw object mask
 bgr_frame = mrcnn.draw_object_mask(bgr_frame)
 # Show depth info of the objects
 Measure =mrcnn.draw_object_info(bgr_frame, depth_frame, depth_frame_rs,Intrinsic)
```

The code reported above starts the Realsense camera streaming (`rs.get_frame_stream()`) and the object recognition (`mrcnn.detect_objects_mask(bgr_frame)`, `bgr_frame = mrcnn. draw_object_mask(bgr_frame)` and `Measure =mrcnn.draw_object_info(bgr_frame, depth_frame, depth_frame_rs,Intrinsic)`).
Once one object is found, its X-Y-Z position is measured 3 times.

```
pub=rospy.Publisher('ObjectPose',Pose,queue_size=1)
pub.publish(estimated_pose)
```

The mean value of these measurements is passed to a ROS publisher (section 3.1.1). This publisher communicates with a ROS subscriber, which is written in the code to control the robot, giving as topic the three coordinates found.

2.4.2 Measures

To estimate the precision of the previous algorithm some measurements were made.

Environments Setup

The same environmental setup, previously used to perform the **Find-Object** measurements, was used (section 2.2.1). The test bench is shown in figure 2.19. This time, 4 different depth distances were considered. They represent, considering the camera reference system, the Z-coordinate of the object. The distances are 310 mm, 370 mm, 410 mm, 460 mm. For each of these Z coordinates, 10 different X positions were measured. These position do not vary with respect to the Z-coordinate. Table 2.3 reports these X positions. For each of these points, 100 measurements were taken. To keep track of them, they were written in .txt files.

Table 2.3: X-coordinates

X position [mm]
-100
-70
-50
-30
0
30
50
70
100

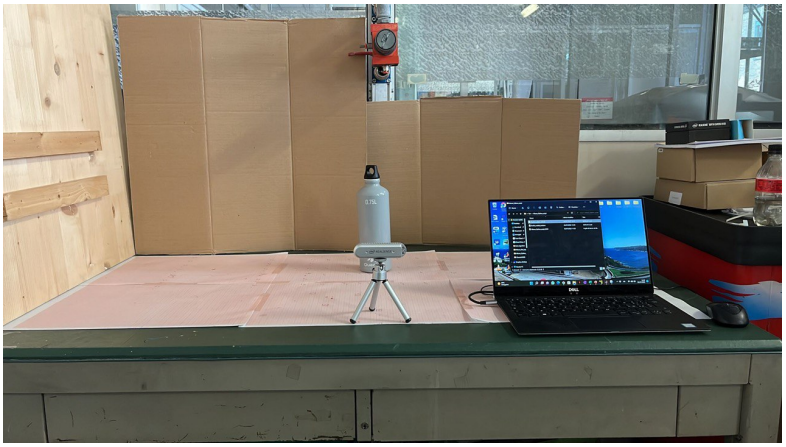

Figure 2.19: Environment Setup

Table 2.4 is an example of the stored measurements taken during the experiments. They were used to estimate the accuracy of this solution and to draw Matlab graphs.

Table 2.4: Measurements Table

X measured [mm]	Y measured [mm]	Z measured [mm]
6.8	-6.8	308
4.5	-7.3	308
6.5	-6.3	308
5.1	-7.1	308
6.2	-6.4	308
5.5	-7.2	308
6.1	-7.7	308
6.8	-6.7	308
6.8	-6.7	308
5.5	-7.1	308
5.8	-6.5	308
6.2	-6.7	308
6.0	-6.8	308
5.9	-6.5	308
5.7	-7.0	308

Matlab graphs were used to better understand the algorithm used to estimate the position of the object in space. For each of the listed positions on the Z axis, three graphs (one for X, one for Y and one for Z) were drawn with the real value and the resulting mean, maximum and minimum value. The figures 2.20 2.21 and 2.22 represent the X, Y and Z measurements in relation to the Z equal to 310 mm

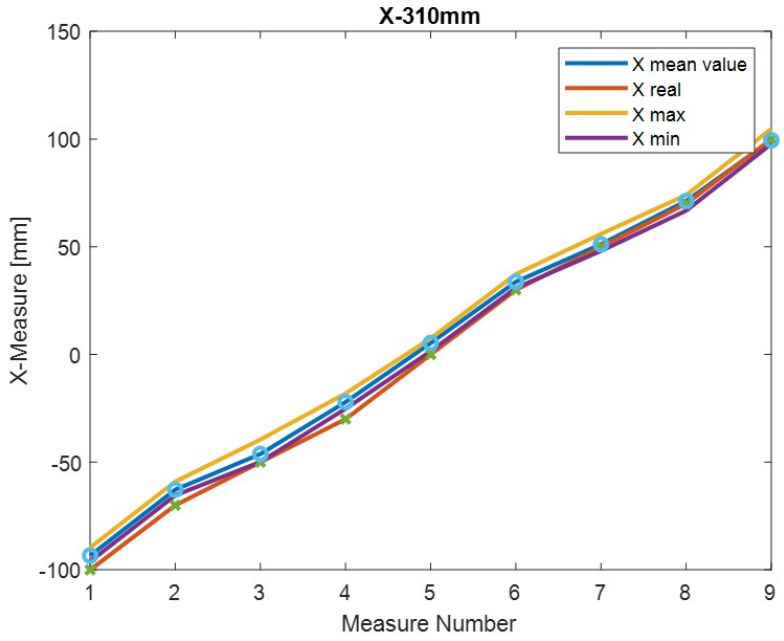

Figure 2.20: X-310

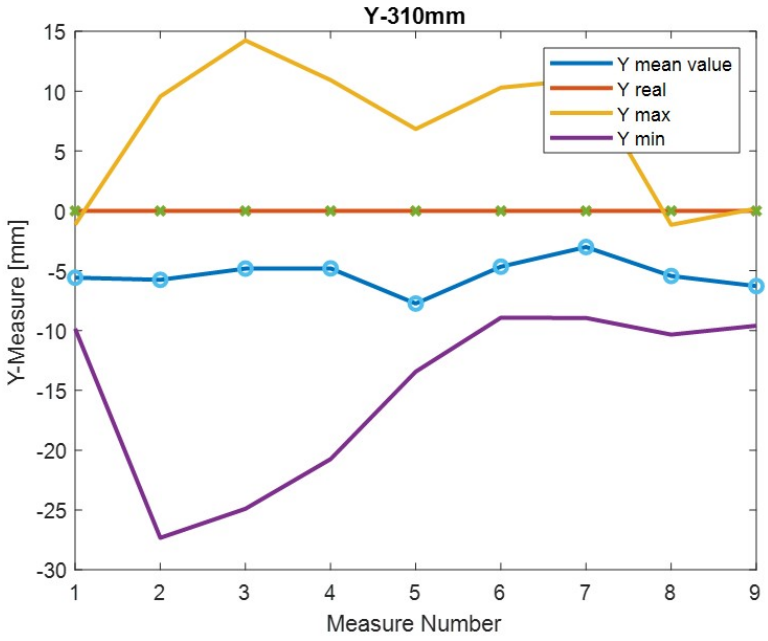

Figure 2.21: Y-310

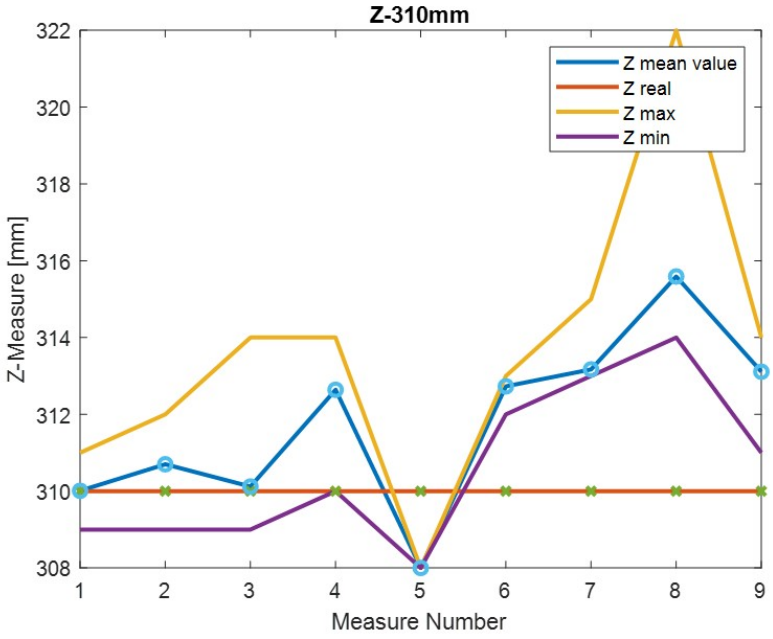

Figure 2.22: Z-310

As it is possible to see from the three graphs in figures 2.20 2.21 2.22 shown above the results are quite good. The first graph X-310, displayed in figure 2.20, shows that the mean values calculated with the algorithm are really close to the real values. In this case, there is no drift at all in any position, even when the object has been moved left and right in relation to the camera. So there is no weak point, as for the **Find-Object** algorithm. Concerning the Y values 2.21, it is clear that the mean values are constant around −5 mm. These values can be considered acceptable for the aim of this work. Figure 2.22, displays the Z values at 310 mm. It is possible to notice that, once more the mean values are not far from the real ones and, the maximum error between the two values is around 5 mm.

To be sure that the depth distance does not negatively influence the measurements, it is possible to analyze also a bigger distance between the camera and the object. Down here figure 2.23, figure 2.24 and figure 2.25 represent the measurements with a Z value equal to 410 mm.

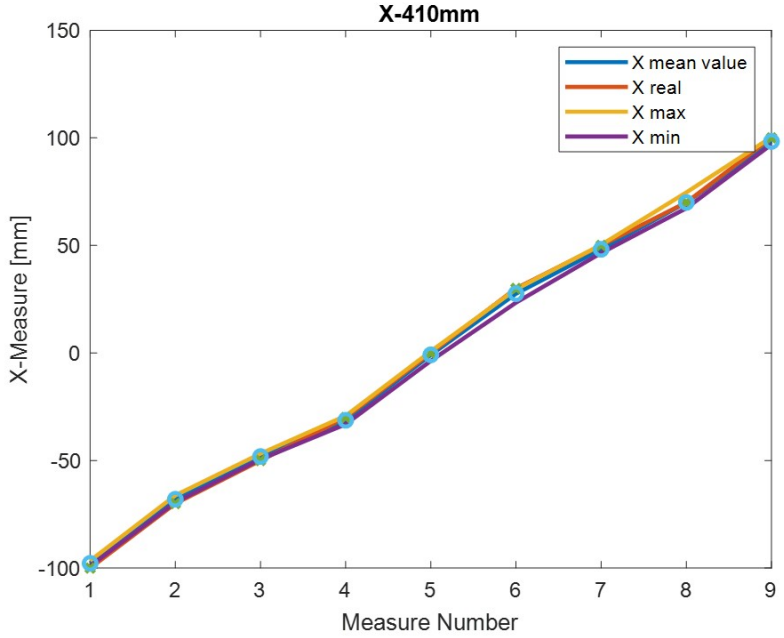

Figure 2.23: X-410

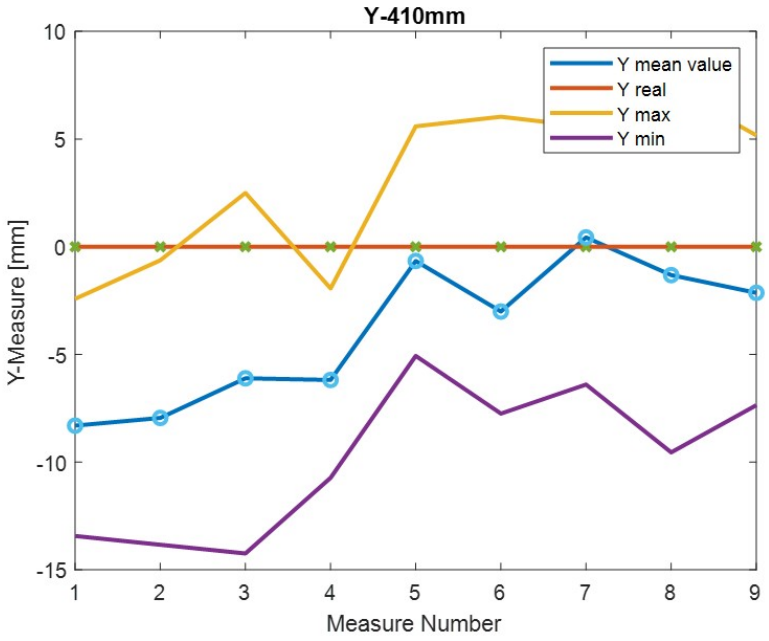

Figure 2.24: Y-410

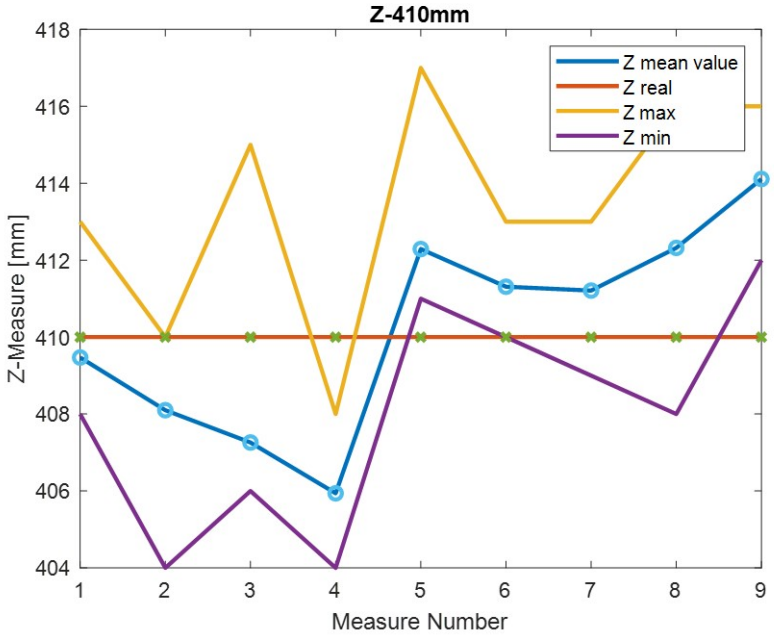

Figure 2.25: Z-410

The results are similar when comparing the diagrams of Z-310 mm and Z-410 mm. So there is no loss of precision even if the distance of the camera to the object increases.

At last, it is also interesting to analyse the overall result. For this purpose, two surfaces of the mean error on the X and Z axis were drawn. Figure 2.26 and figure 2.27 show the results obtained.

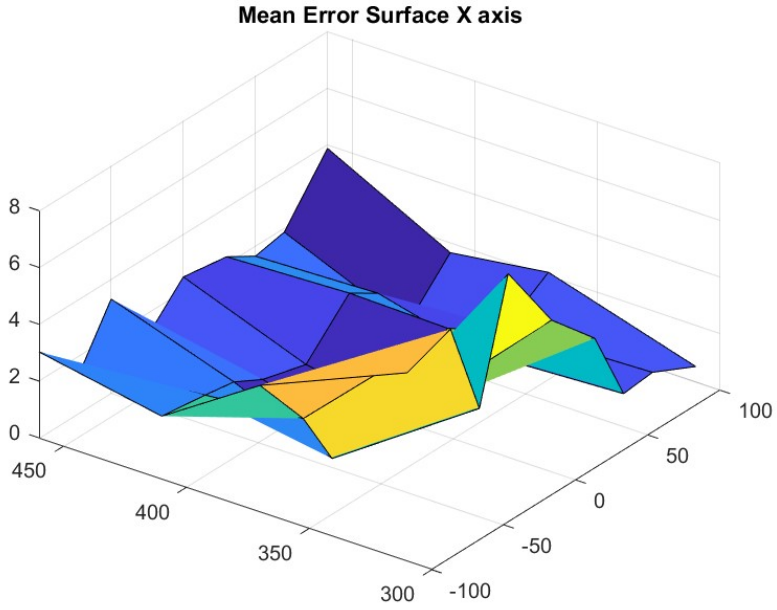

Figure 2.26: Mean Error Surface X axis

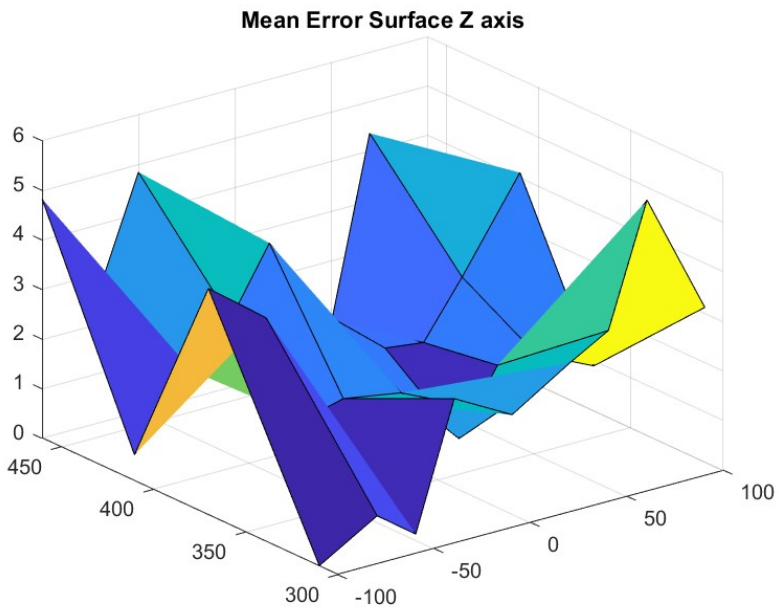

Figure 2.27: Mean Error Surface Z axis

From these diagrams it is once again clear that this way of estimating the X-Y-Z positions of the object in relation to the camera is good enough to be used. There is no mean error greater than 8 mm for the X-axis and 5 mm for the Z-axis.
After all the experiments on the algorithms and the precision of the camera were completed, the work moved to a simulation environment.

Chapter 3

Virtual Environment Simulation

This chapter is about the simulation, which was carried out in the virtual environment GazeboSim. The first is an introduction to the ROS framework; the second is a section on the simulated hardware; the third section is about the ROS packages and toolbox used to control the simulated robot. The last section deals with the simulation performed and the code used to move the robot.

3.1 ROS

Robot Operating System (ROS) is a flexible framework that gives various tools and libraries for writing robotic software. It provides several features to help developers in activities such as message transfer, distributed computing, code reuse, and implementation of state-of-the-art algorithms for robotic applications. There are a variety of benefits in using ROS as a programming framework [26]:

- High-end capabilities: ROS has functions that are ready to use. For instance, the MoveIt package in ROS may be used for motion planning for robot manipulators, while the Simultaneous Localization and Mapping (SLAM) and Adaptive Monte Carlo Localization (AMCL) packages in ROS can be used for autonomous navigation in mobile robots.
- Tons of tools: ROS ecosystem is full of tools for debugging, visualization

and simulation. Some of the most effective open source tools are rqt gui, RViz, and Gazebo.

- Support for high-end sensors and actuators: it is possible to use numerous device drivers and interface packages for various sensors and actuators in robotics thanks to ROS.
- Inter-platform operability: different applications can communicate with one another thanks to the ROS message-passing middleware.
- Modularity: the system can still work even if one node crashes.
- Concurrent resource handling: using ROS it is possible to reduce complexity in computation.

3.1.1 ROS Computation Graph

A network of ROS nodes is used to do computations in ROS [26]. The computation graph refers to this network of computation. The ROS nodes, master, parameter server, messages, topics, services, and bags are the core ideas in the computation graph. This graph is shown in figure 3.1.

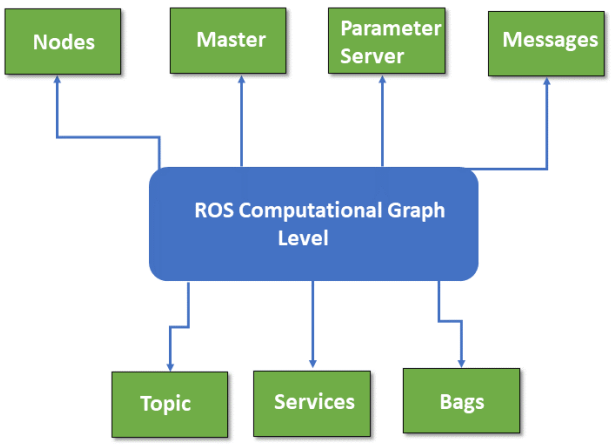

Figure 3.1: Structure of the ROS graph layer

A small explanation of the ROS computation graph is given below[26].

- Nodes: the processes that do computing are called nodes.
- Master: the nodes name registration and search procedures are handled by the ROS master. Without a ROS master, nodes won't be able to locate one another, communicate with one another, or use services.
- Parameter server: it allows to store data. These values are accessible and modifiable by all nodes. The ROS master includes the parameter server.
- Topics: in ROS, each communication is delivered using a named bus called topic. A node is publishing a topic when it sends a message to another node. A node is said to be subscribing to a topic when it subscribes to a topic through which it gets messages.
- Logging: logging mechanism is offered by ROS to save data. They are referred to as bagfiles. When working with intricate robot mechanics, bagfiles are a very helpful tool.

The graph, shown in figure 3.2, displays the topic-based communication between the nodes:

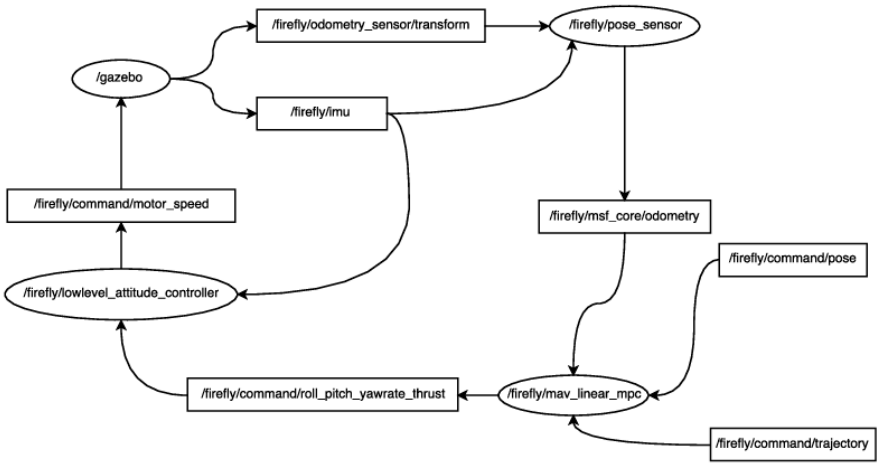

Figure 3.2: Graph of communication between nodes using topics

Rectangles are used to represent topics, and ellipses are used to represent nodes.

Publisher/Subscriber

The Publisher Subscriber Interface provided by the functions of the ROS library is the way messages are passed in ROS [27]. A ROS node can be both a subscriber and a publisher. A publisher puts the messages of some standard message type to a particular topic. On the other hand, the subscriber subscribes to the topic so that it may get notified whenever a message is posted to the topic. A publisher can to publish more than one topic and a subscriber may also receive more than one topic. In addition, neither the publisher nor the subscriber knows of each other's existence. The goal is to separate the creation and consumption of information, and the ROS master keeps track of all the nodes IP addresses. The explanation given up here is nicely summarized in figure 3.3. From it, it is possible to see how, the node on the left (the publisher) sends a topic and the nodes on the right (the subscribers) can receive it.

Figure 3.3: Publisher Subscriber Model

3.2 UR5

3.2.1 UR5 Description

The robot used for the simulation was the UR5. The extremely adaptable Universal Robots UR5 robotic arm makes it possible to safely automate hazardous or repetitive jobs [28]. It is the ideal cobot for carrying out minor activities like packaging, assembling, or testing because it has a carrying capacity of 5 kg and an operating radius of 850 mm. The UR5 is incredibly simple to set up. A touchscreen tablet may be used to further tune the robot once it has been manually moved to the proper places.

The robot is composed of 6 rotational joints which allow the robot to have 6 DOF. Each joint can rotate of $\pm 360°$ and can reach a maximum speed of $\pm 180°/s$. The robot is almost completely made of steel besides the junctions between links which are covered with PP plastic. The previously described UR5 robot can be see in figure 3.4.

Figure 3.4: UR5 Robot

3.2.2 UR5 programming

The Universal Robot family of robots has a patented programming interface called PolyScope. Using that interface the company claims that it is easy to program the robot to move along the desired trajectory [29]. The majority of the tasks can be completed by programming on the teach pendant. Figure 3.5 shows the teach pendant for the UR5 . It is a touchscreen tablet where all the programming interface is displayed. Using it, it is possible to control the robot and set safety parameters. The teach pendant has on one side an emergency button, useful in a hazardous situation, and also the power button.
Teaching a robot how to move is critical because tool motion is an integral

part of a robot program. For this reason, a process must be developed to teach the robot to move in an appropriate manner. In PolyScope, movements of the tool are specified using a list of waypoints, or locations inside the workspace of the robot. A waypoint can be indicated by putting the robot in a certain place or by using software to compute it. In order to move the robot arm to a certain position the Move tab, in figure 3.5 the display interface shows it, can be used. It is also possible to just drag the robot arm into position while holding the teach button on the back of the teach pendant. Besides controlling the robot through the pendant, it is also possible to send I/O from other devices. The technique of sending input command from a different device is going to be explained in the following chapter.

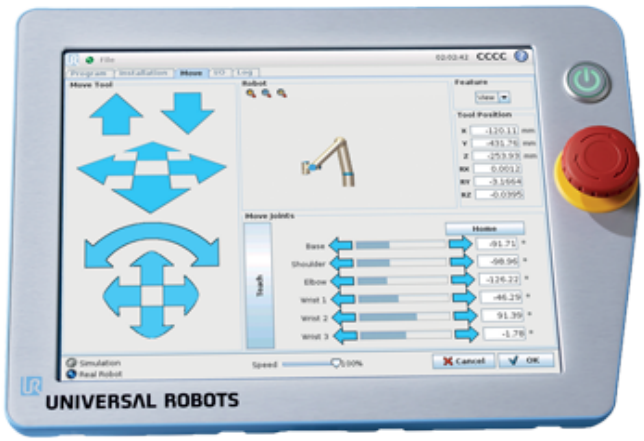

Figure 3.5: UR teach Pendant displaying Move Tab

3.2.3 Overview of Client Interfaces

It is worth specifying the communication interfaces which is possible to use in the UR5. The UR robot can communicate with outside equipment via a variety of interfaces [30]:

- Primary/Secondary Interfaces: to communicate robot state data and receive URScript commands, UR controller servers are available. Robot state information and other communications are sent through the main interface. Only robot state data are transmitted through the secondary interface. The data are mostly utilized for the controller and Graphical User Interface (GUI) communication. Both accept 10 Hz update rate URScript instructions. It enables remote robot control.

- The Real-time Interfaces: real-time interfaces perform similarly to primary and secondary interfaces. The controller communicates information about the robot state and receives URScript instructions. The update rate is the primary distinction. Real-time interface sends information at a rate of 125 Hz.

- Dashboard Server: sending straightforward commands to the GUI through a TCP/IP[1] connection enables remote control of a Universal Robot. "Dashboard Server" is the name of this interface. The server primary responsibilities include receiving feedback on robot condition and setting user access levels. It may also load, run, pause, and stop robot programs.

- Socket Communication: it may be used by the UR robot to connect with external devices. Data may be transmitted between a robot and another device via a socket communication. The Robot acts as a client and other devices play the role of server in socket communication. URScript has instructions that enable socket opening and closing as well as the sending and receiving of various data types.

- XML-RPC: XML-RPC: it is a Remote Procedure Call technique that transfers data between applications over sockets using Extensible Markup

[1]TCP/IP Protocol is developed in the next chapter section 4.2

Language (XML)[2]. Using this standard the UR controller is able to call methods/functions from a different server/program and have as response structured data. It may be used to execute a sophisticated computation that is not possible using URScript.

- RTDE (Real-Time Data Exchange): it is a substitution for real-time interface. RTDE enables the UR controller to send custom state data and accept custom set-points and register data.

Figure 3.6 is a good summary for what has been said before about robot communication interfaces.

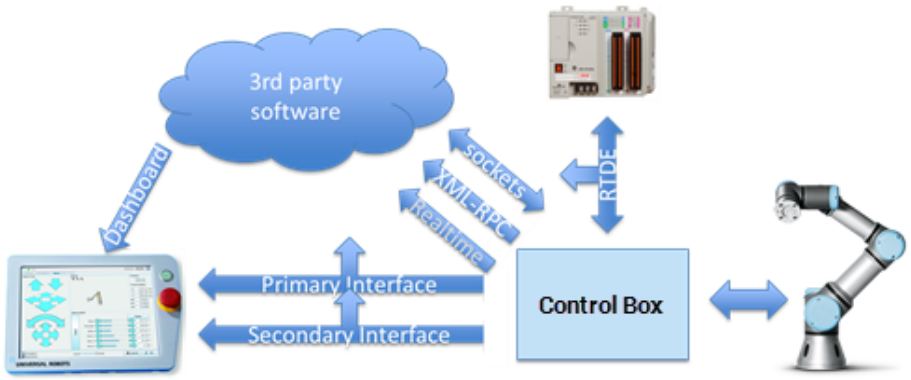

Figure 3.6: Communication Interfaces

One thing to point out is that, in the previous list, the TCP/IP protocol has been appointed many times without giving a proper description and interpretation to it. This is going to be done in the following chapter together with the explanation of the used client interface.

[2]A markup language and file format for storing, sending, and recreating arbitrary data is called Extensible Markup Language (XML)

3.3 MoveIt!

The ability to transition from an initial position to an assigned final position is the minimum criterion for a manipulator [26].
To have full control of a robot, the problem of motion planning must be solved. The goal, in this case, is to create a trajectory by simply setting some inputs and constraints for the motion controller. Controlling each joint and manually calculating its motion could be a difficult task. Therefore, a set of packages and tools, for doing mobile manipulation in ROS, called MoveIt!, were used.
MoveIT! contains a set of software for motion planning, manipulation, three-dimensional (3D) perception, collision detection and navigation. It includes some graphical user interfaces for a new robot configuration as well as the possibility to perform motion planning either via a User Interface (UI) or via Application Programme Interfaces (APIs).

3.3.1 MoveIt! structure

MoveIt! has a peculiar high-level architecture with a primary node in the middle (move_group), which is basically a bridge between the user and the robot, and all the other nodes around it. Figure 3.7 is an overview of the MoveIt! architecture. As it is possible to see, the move_group performes the interation role. It combines all the different parts coming from different sources and provide the user a selection of ROS action and services.

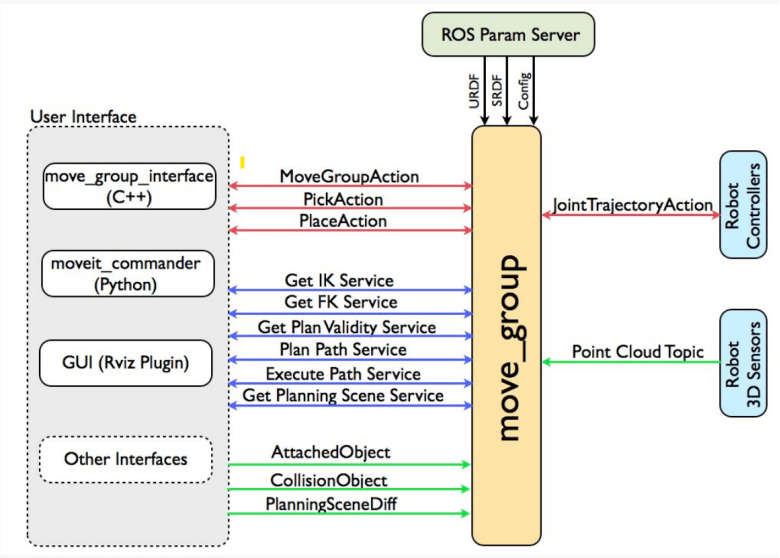

Figure 3.7: MoveIt! architecture

From figure 3.7, it is evident that the move_group node gathers all the robot data in form of topics and services, including joint state and transformations. In the ROS Param Server, after the generation of the MoveIt! package, all the kinematics information of the manipulator are stored. This information is:

- Unified Robot Description Format (URDF): this is an XML format file for representing a robot model.
- Semantic Robot Description Format (SRDF): a representation of semantic information about robots.
- Configuration Files

These data are exchanged with the move_group.
Once MoveIt! has collected all the necessary data about the robot and its setting, it is possible to control it. The C++ or Python Moveit! APIs can be utilized to carry out tasks. Also, the RViz[3] motion planning plugin, which allows to command the robot from the RViz GUI, is a possible alternative.

[3]RViz is a 3D visualizer for the Robot Operating System (ROS) framework

The move_group can be considered a link between the user and the robot, therefore it does not run any motion-planning algorithms directly. To do so, MoveIt! interacts with plugins for motion planning, kinematic calculation and other functions. The robot controller, after the motion planning, communicates with the robot using an action interface called *FollowJointTrajectoryAction*.

3.4 Configuration's packages using Setup Assistant tool

In order to configure the robot to MoveIt!, the built in Setup Assistant Tool was used. It is a user friendly GUI which generates SRDF, configuration files and some code used by the move_group node.

The SFDF file provides information on the virtual joints, collision-link pairs, and end effector joints that are configured using the Setup Assistant Tool. The configuration files store all the information about the kinematic solver, joint limits, controllers, etc. Launch files are used to start all the ROS nodes necessary to run the simulation and mainly start the move_group and the selected controller.

3.4.1 Create a New MoveIt Configuration Package

The following steps are the ones used to create the Configuration Package throughout the MoveIt! Setup Assistant [26] [31] :

1. use the command: **roslaunch moveit_setup_assistant setup _assistant.launch** to open the MoveIt! Setup Assistant

   As shown in figure 3.8, it is possible to choose between the creation of a new MoveIt! Configuration Package or edit an existing one. Create New MoveIt Configuration Package was selected. The creation of a new package requires to select a *urdf* file called */universal_robot/ur_description/urdf/ ur5_robotiq85_gripper.urdf.xacro*. This *urdf* file contains the description of the UR5 robot, with a robotiq 2f-85 gripper, attached to its end effector. If the loading of the file was successful, the windows in figure 3.9 should appear.

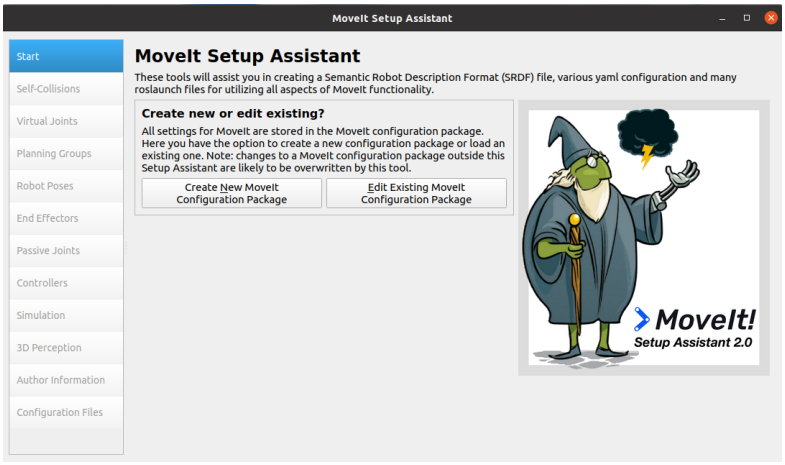

Figure 3.8: MoveIt! Setup Assistant

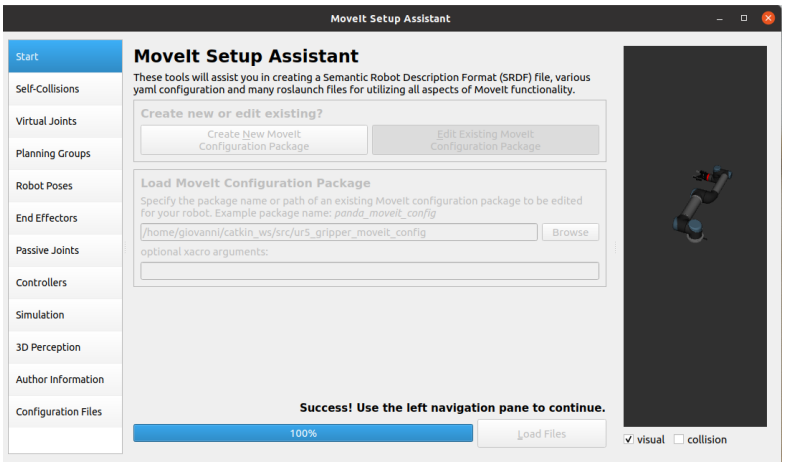

Figure 3.9: Successfully loaded file

On the right side of figure 3.9 a visualization of the robot's model with the end-effector connected is shown.

2. The second step was to set the self-collision. This was possible by pushing the *Generate Collision Matrix* button. What should appear is a window showing the Self-Collision matrix, as shown in figure 3.10.

   This step speeds up the processing. Each link pair is examined by this tool, which classifies each link pair as either always colliding, never colliding, colliding in the robot's default position, with neighbouring links disabled, or occasionally colliding.

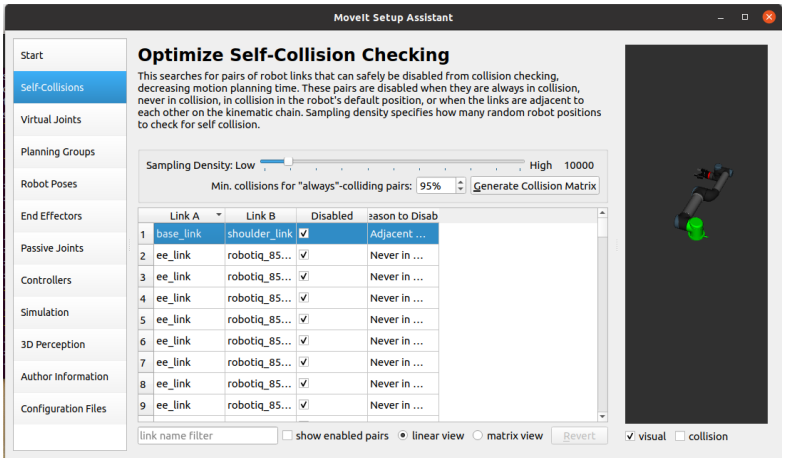

Figure 3.10: Self-Collision

3. The third step was the definition of Virtual Joints used to create a connection between the robot link and an external frame of reference. Figure 3.11 shows that only one virtual joint was created. This joint was called *Virtual_joint* and it has as a parent frame the world frame and as child link the base_link of the UR5 robot.

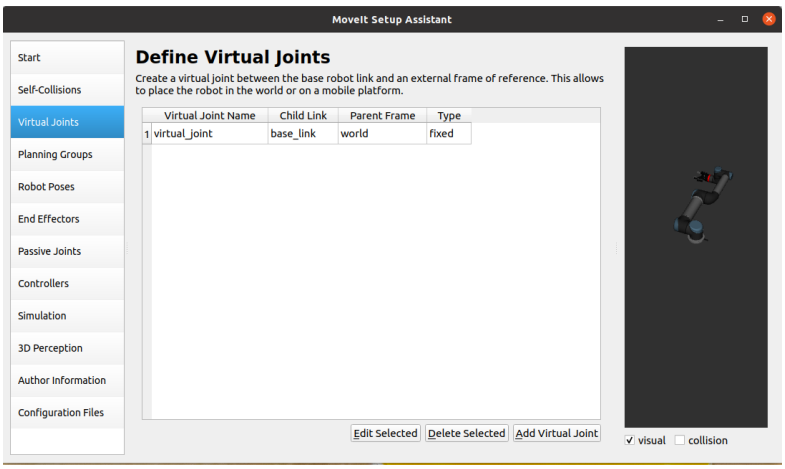

Figure 3.11: Virtual Joints

4. The definition of the planning group allowed MoveIt! to create a group of links/joints, of the robotic arm, to be considered unique, when planning a goal position, either for the end effector or for a link. Two different planning groups were defined in this work. One was called *ur5_arm*

and the other *gripper*. They correspond to the chain of links from the base link of the UR5, to the end-effector link and to the robotiq gripper, respectively. Everything stated above is shown figure 3.12, where both the *ur5_arm* group and the *gripper* group are defined.

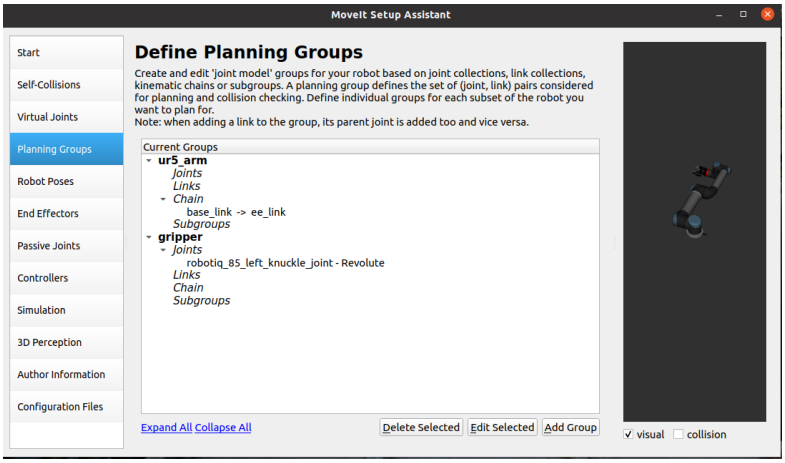

Figure 3.12: Planning Groups

- ur5_arm: in the creation of the moving group the Kinematic Solver and the type of Planning were selected.
  Figure 3.13 shows the window interface where to select them.

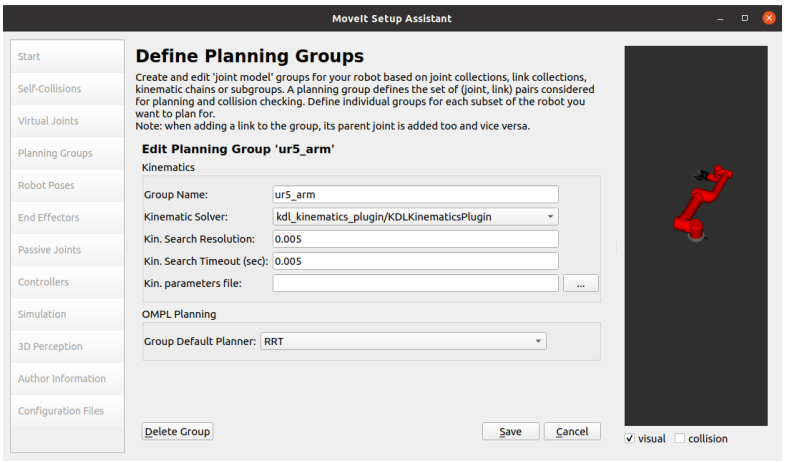

Figure 3.13: ur5_group

The kinematic solver is responsible for the calculation of all the inverse kinematics. The default solver, called *kdl_kinematics_plugin/*

KDLKinematicsPlugin was selected. This solver is provided by the Oroscos KDL package and adheres the limitation outlined in the URDF. The second setting to select is the Planner for the Planning Group. The default planner in Moveit is the Open Motion Planning Library (OMPL). For an optimization goal, there are particular planners in the OMPL planning library that can be optimal for that goal. Also in this case the suggested planner called rapidly exploring tree (RRT), was the chosen one. In order to solve geometric planning the RRT generates a rapidly exploring tree (RRT). Therefore RRT is a tree-based motion planner which constructs a search tree iteratively using samples chosen at random from the state space.

5. The next step was the definition of robot poses, which include sets of joint values for certain planning groups.

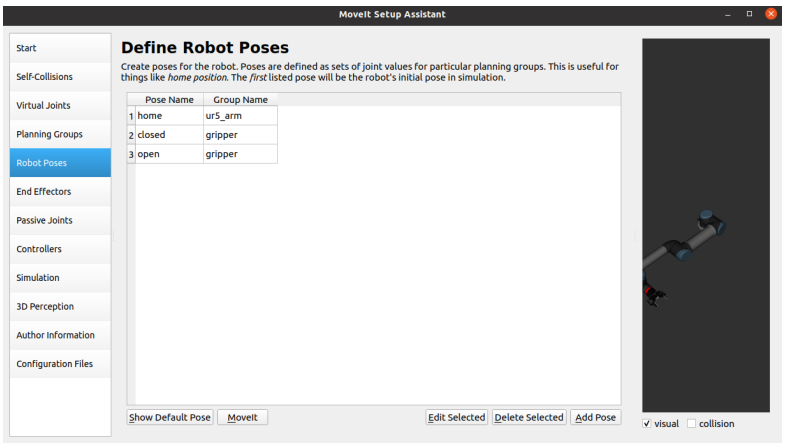

Figure 3.14: Robot Poses

From figure 3.14, it is possible to see that 3 different poses were defined. One for the *ur5_arm* group:

- home: the home position[4] of the UR5 arm. Figure 3.15 shows the default values for the robot's joints.

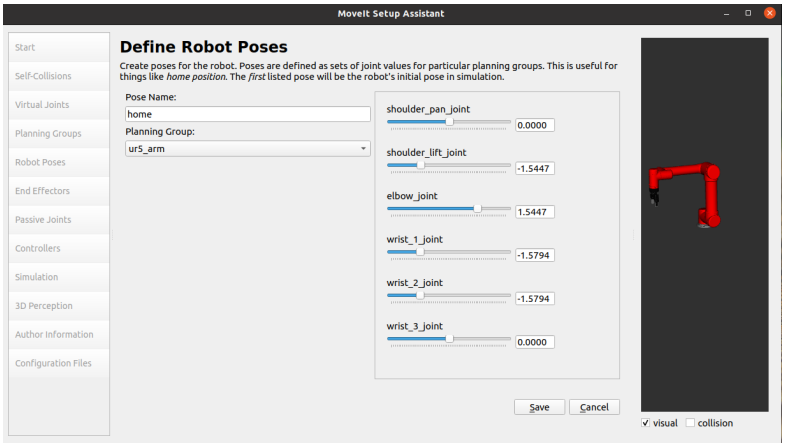

Figure 3.15: Home Position

Two for the *gripper* group:

- open: which defines the gripper as open, shown in figure 3.16.

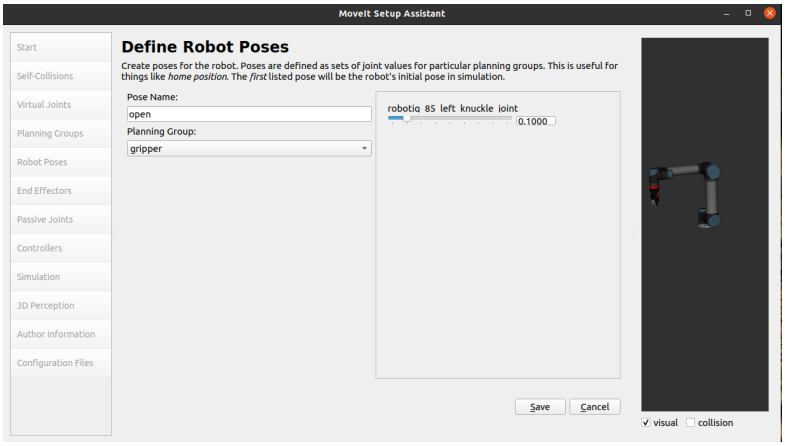

Figure 3.16: Open gripper

- closed: which defines the gripper as close, shown in figure 3.17 .

[4]The starting position of the robot

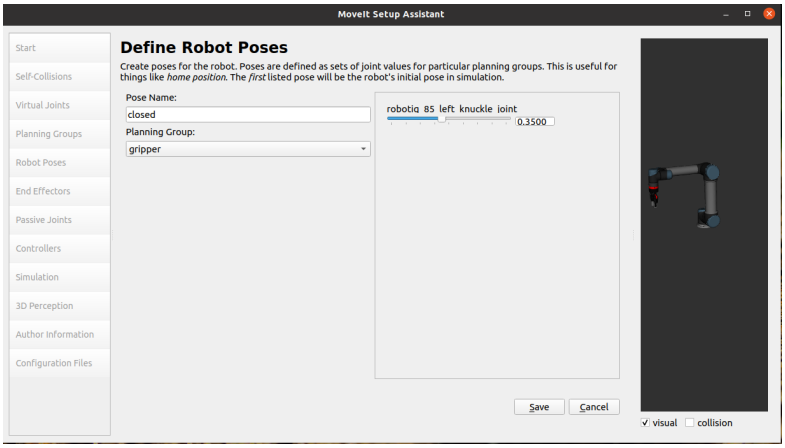

Figure 3.17: Closed gripper

6. The last two steps were the definition of the end effector and of the passive joints, which are the joints that move only passively by the movement of the other joints.

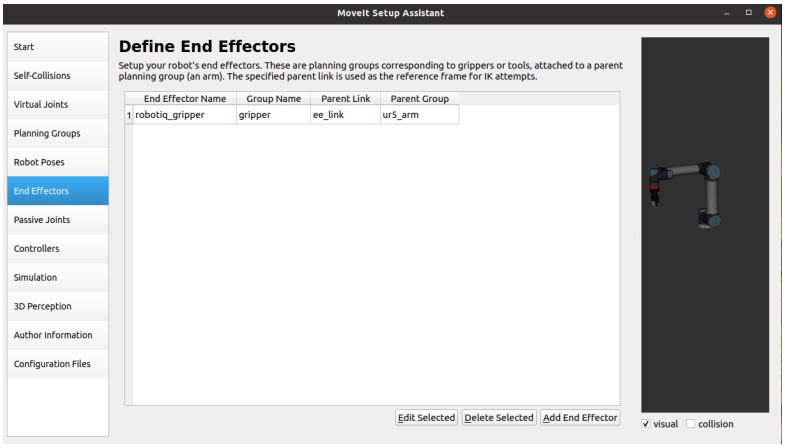

Figure 3.18: End Effector

In the first passage, the *gripper* group was set as the End effector. This is illustrated in figure 3.18. Then, concerning the Passive Joints some parts of the gripper, which are not responsible for the motion, were selected as passive parts, this is shown in figure 3.19.

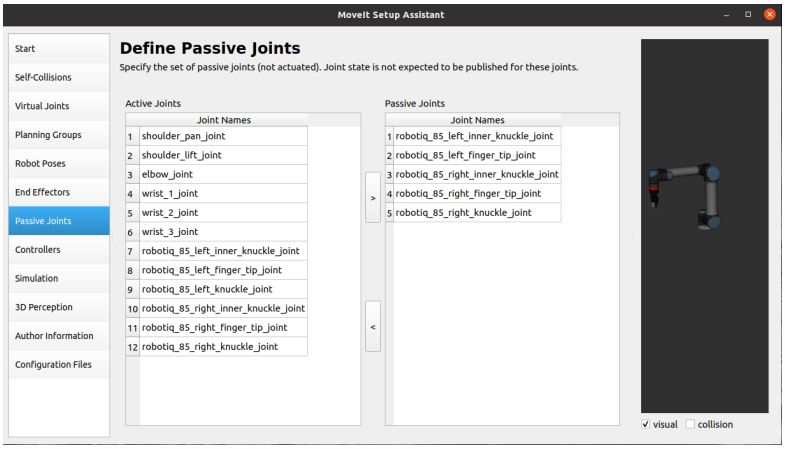

Figure 3.19: Passive Joints

After this last step, *MoveIt!* was ready to generate the package **ur5_gripper_moveit_config**. This folder contains all the configurations and launch files needed to simulate and control the robot.

3.5 Gazebo Simulation

After the previous settings regarding robot control were made, grasping with the UR5 on Gazebo can be simulated. Gazebo is a dynamic 3D simulator that can efficiently and accurately model robots both indoor and outdoor. Gazebo includes a physics simulation with a much higher level of detail, a collection of sensors and interfaces for both users and programs [32].

The Gazebo environment consisted of the UR5 robot on one table, another table in front of the robot on which the object to be grasped was placed, and finally, on one side of the second table, a camera model used for object recognition and 3D position estimation of the object.

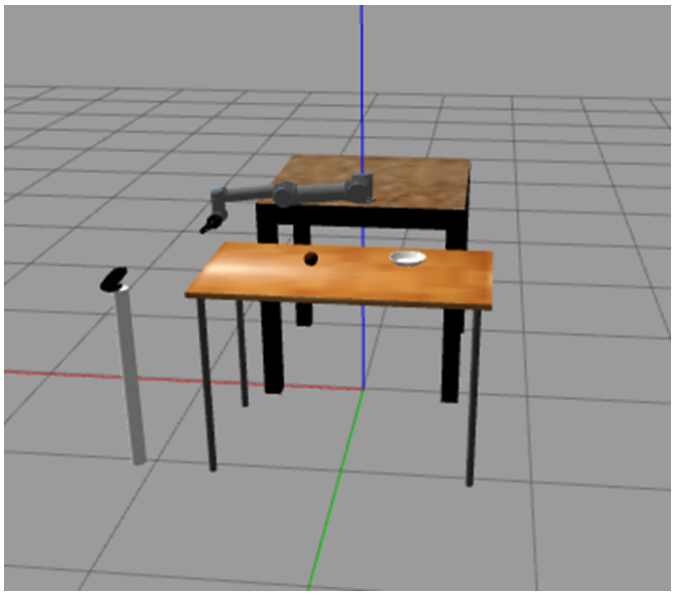

Figure 3.20: Gazebo Environment

At this point, the robot is ready to be moved. A python code [33] was written to move the robot and place it in the correct position for grasping.

```
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('move_group_python_interface', anonymous=True)

robot = moveit_commander.RobotCommander()
scene = moveit_commander.PlanningSceneInterface()

arm_group = moveit_commander.MoveGroupCommander("ur5_arm")

hand_group = moveit_commander.MoveGroupCommander("gripper")
```

The main user interface in MoveIt! is provided by the `RobotCommander` class. With `moveit_commander.roscpp_initialize(sys.argv)` the above class is initialized. Then, a ROS node is created by writing `rospy.init_node('move_group_python_interface', ...)`. The next two passages are the creation of a `RobotCommander` object, which serves as an interface to the robot as a whole, and of a
`PlanningSceneInterface` object, which is required to communicate with the world around the robot. Next, two `MoveGroupCommander` objects are instantiated (`arm_group` and `hand_group`). These objects provide an interface to the two joint groups .

The groups in this case consist of the joints of the UR5 arm and the joint of the gripper. The movements of the UR5 arm and the gripper can be planned and executed via these interfaces.

After the initialization phase, the planning is ready to be started.
The grasping procedure has been organized in 4 consecutive steps:

1. Home Position: the robotic arm is set in a predefined position and the gripper is opened.
2. Approaching Position: the robotic arm is moved in a position which is close to the object.
3. Grasping Position: the robot is moved in the position for the grasping and the gripper is closed.
4. Final Position: the robot is moved to a final position.

To put the arm in the start position, a method of the `MoveGroupCommander` object is used.

```
arm_group.set_named_target("home")
plan1 = arm_group.go()
plan1 = arm_group.stop()
```

Using the method `arm_group.set_named_target("home")` the robot moves to the home position. Once the robot is in the home position, the gripper can be opened. A similar procedure is used and a `hand_group.set_named_target("open")` method is called. Figure 3.21 shows the robot, in the simulation environment, set in the home position.

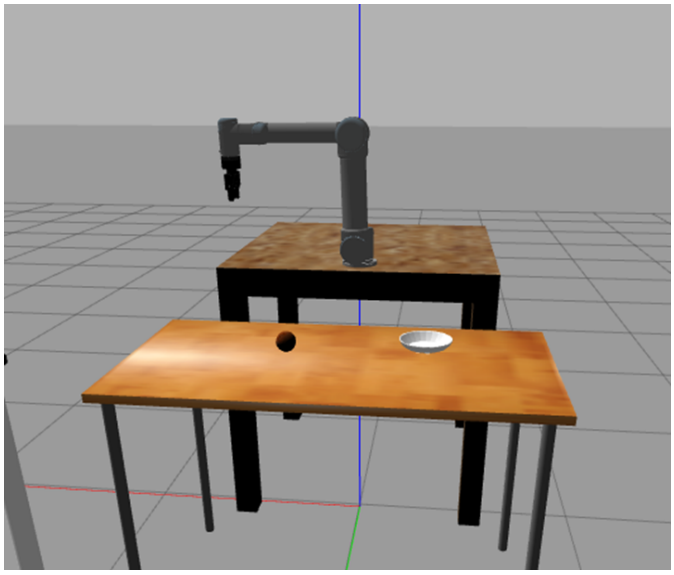

Figure 3.21: Home position

In the next step, the robot waits for the object pose to come from the camera. The camera code publishes the found XYZ coordinates as ROS topic. Then, they are taken and converted, from the camera reference frame to the world one. Figure 3.22 shows the two reference frames.

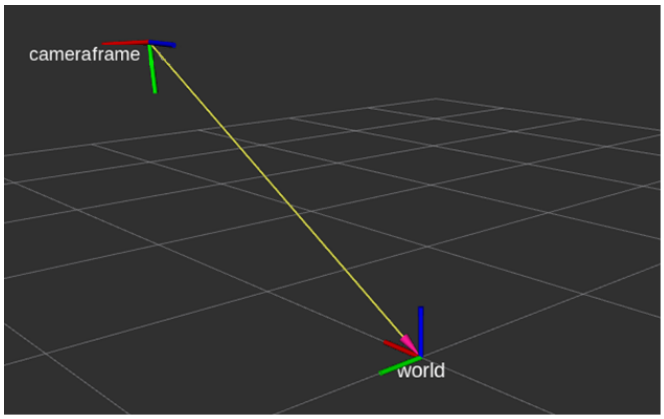

Figure 3.22: World and Camera Reference Frames

```
print("Waiting for object pose...")
pose_sub=rospy.wait_for_message("/ObjectPose",Pose)
print("Object pose found!")
#Rotation Matrix to convert the position from camera to world
    frame
Rotation_matrix=np.array([[0,0,-1],[1,0,0],[0,-1,0]])
#Converting the position to a suitable value for the simultion
Camera_x=round(pose_sub.position.x/1000,2)
Camera_y=round(pose_sub.position.y/1000,2)
Camera_z=round(pose_sub.position.z/1000,2)
#Dinstance between the world and the camera frame
Frame_dinst=np.array([1.1,1,1.04])
Point=np.array([Camera_x,Camera_y,Camera_z])
#Convertion from the camera to the worls frame
World_position=Rotation_matrix.dot(Point)+Frame_dinst
X_position=round(World_position[0],2)
Y_position=round(World_position[1],2)
Z_position=round(World_position[2],2)-1.13
```

In the `pose_sub` variable, the topic published by the camera node is saved. Since the topic from the publisher is not immediately available, the ROS method *.wait_for_message* is used. This method allows freezing of the robot movement until the `/ObjectPose` topic is not published in ROS network. The received pose is converted to the world frame using the following formula:

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} = \begin{bmatrix} wx_1 & wy_1 & wz_1 \\ wx_2 & wy_2 & wz_2 \\ wx_3 & wy_3 & wz_3 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{3.1}$$

Where the 3x3 matrix composed is the rotation matrix, $X_cY_cZ_c$ are the coordinates in the camera frame and $t_xt_yt_z$ are the distances between the camera and the world frame. These distances are $t_x = 1.1\,\text{m}$ $t_y = 1\,\text{m}$ $t_z = 1.04\,\text{m}$.

Now that X_w Y_w Z_w are available it is possible to move the arm toward the object.

```
pose_target = geometry_msgs.msg.Pose()
pose_target.orientation.w = current_pose.orientation.w
pose_target.orientation.x= current_pose.orientation.x
pose_target.orientation.y = current_pose.orientation.y
pose_target.orientation.z = current_pose.orientation.z
pose_target.position.x = X_position
pose_target.position.y = Y_position
pose_target.position.z = Z_position+0.3
arm_group.set_pose_target(pose_target)

plan1 = arm_group.plan()
plan1 = arm_group.go(wait=True)
```

To do so, the method `arm_group.set_pose_target(pose_target)` is used. It receives as input the target pose which in this case, is the object position, with an offset on the Z axis. The offset is set because, during this stage, the robot has to move in the approaching position. This step was established because it allows the robot to get close to the object. Up to this phase, the robot can move quickly without worrying about precision. Then, in the following phase, the robot can move slower and with higher precision. `arm_group.plan()` and `arm_group.go(wait=True)` are used to make the robot move. Figure 3.23 displays the robotic arm in the resulting position. As it is possible to see, the gripper is close to the object, but has an offset on the Z-axis.

```
pose_target = geometry_msgs.msg.Pose()
pose_target.orientation.w = current_pose.orientation.w
pose_target.orientation.x= current_pose.orientation.x
pose_target.orientation.y = current_pose.orientation.y
pose_target.orientation.z = current_pose.orientation.z
pose_target.position.x = X_position
pose_target.position.y = Y_position
pose_target.position.z = Z_position
arm_group.set_pose_target(pose_target)
plan1 = arm_group.plan()
plan1 = arm_group.go(wait=True)
```

Once the robot is in this position, it is ready to be moved in the *Grasping Position*.
The method `arm_group.set_pose_target(pose_target)` is used to make the robot move. As it is possible to see from the code reported above, the

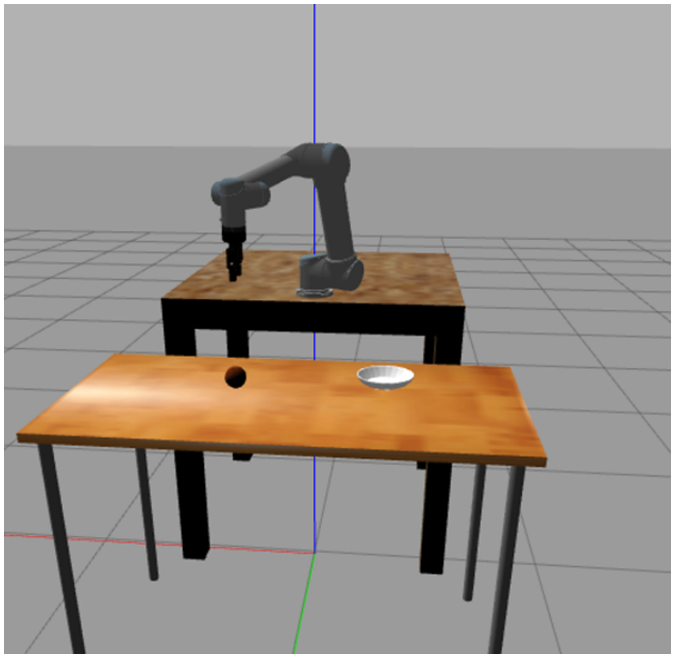

Figure 3.23: Approaching position

only difference with the previous code is that now there is no offset on the Z-axis.

The robot arm moves towards the object and as soon as it is in position, the gripper closes. The closing of the gripper can now take place because the end effector is in a position close enough to make the object reachable for the gripper.

```
hand_group.set_named_target("closed")
plan2 = hand_group.go(wait=True)
```

Once more, the gripper can be simply closed using the two consecutive methods reported above. Figure 3.24, reported below, shows the simulated UR5 in the **Grasping Position**.

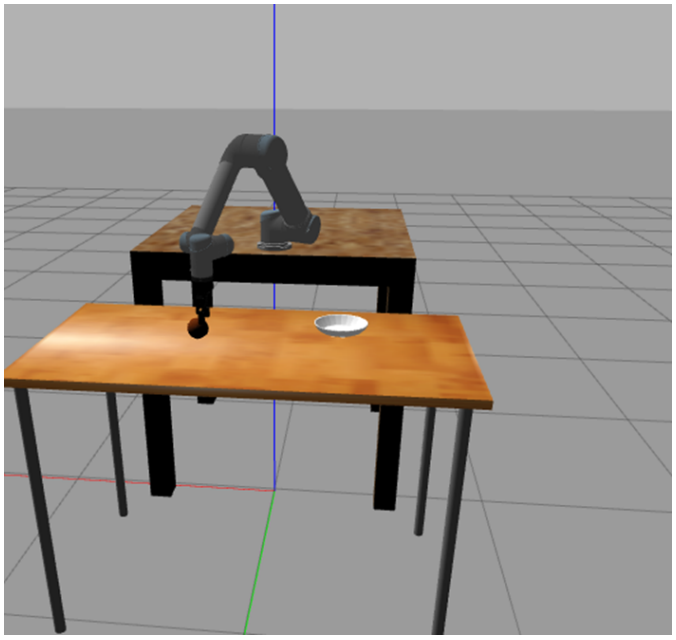

Figure 3.24: Grasping position

In the last step, the object is taken and the arm can be moved to any position. Once again the arm is moved using the **arm_group.set_pose_target(pose_target)** method.

Chapter 4

Real World Simulation

This chapter describes the tests carried out in real world. Both the real robot and the real camera were used to perform the experiments.

4.1 Experimental Set-Up

The experimental set-up was carried out in the DIMEAS laboratory of the Politecnico di Torino. The UR5 is mounted on a trolley, on which the control unit and the teach pendant are mounted, too. The robot working area was limited so that it neither collides with the trolley itself nor goes beyond certain limits. The Realsense camera was placed on one side of the moving trolley, facing the trolley itself. This position was chosen to have enough space for the object to be detected and to ensure that the robot does not interfere with the camera acquisition. On the right side of the moving cart, a table was placed with a PC for controlling the experimental setup and an external monitor. In addition, a Wi-Fi router is used for the connection between the PC and the robot. Figure 4.1 shows the previously described experimental set-up.

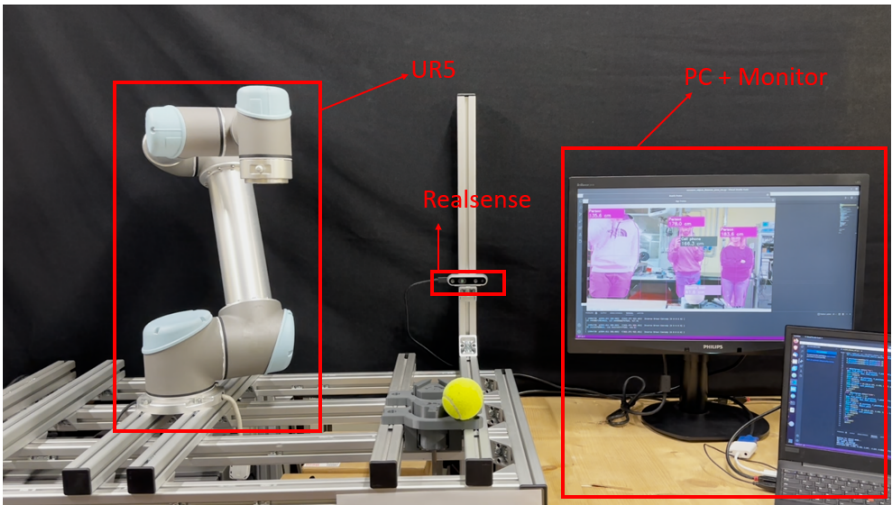

Figure 4.1: Experimental Set-Up

From this figure, it is possible to see that the UR5 robot does not have the gripper. At this stage, a real gripper was not yet available, so all experiments were performed without the gripper.

4.2 Socket Communication

As said in section 3.2.3, the UR robot can communicate with outside equipment via a variety of interfaces. Among those listed in 3.2.3 a Socket communication was chosen. This type of communication uses a TCP/IP [34] [35] [36] protocol to connect with the device.

4.2.1 TCP/IP Protocol

TCP/IP protocol is not bounded by the only TCP and IP protocols, but it is also made of some other protocols that are divide in 4 different layers.

- The network interface layer: it is in charge of transferring frames of data between hosts connected by the same physical network.

- The internet layer: the primary duty of this layer is to route packets from one host to another. Since the data have not yet been organized into a frame for transmission at this level, the focus is on "packets"

rather than "frames"[1]. Each packet carries the address data required for its Internet-wide routing to the destination host. The Internet protocol, or IP (as in TCP/IP), is the dominant protocol at this level.

- The host-to-host layer: regardless of the path or distance utilized to deliver the message, this layer is largely in charge of ensuring data integrity between the sender host and receiver host. At this level, communication faults are found and fixed.

- The process and application layer: this layer offers access to the TCP/IP stack for the user or application programs.

4.2.2 Internet Protocol Version 4

The TCP/IP suite central component is the Internet protocol (IP). From router to router, it is essentially in charge of routing packets to their destinations. The basis for this routing is the IP addresses that are included in each packet header forwarded by IP. Version 4 (IPv4) of the protocol, which employs a 32-bit address, is the most widely used version of IP today. The main characteristic of the IP address is that it does not belong to the node, as the MAC or hardware address does[2], but rather it indicated the place where the connection happened.
As it was said before the IPv4 address is composed of 32 bits.
This number is normally divided into four pieces of 1 byte, which are indicated as a,b,c,d or w,x,y,z. Figure 4.2 shows the 4 bytes divided into w,x,y,z and the binary conversion. The result of this conversion is 192.100.100.1.

From an IP address, two parts can be distinct. The network ID (NetID) is the first component and it is composed of the first 3 numbers of the IP address. It is a special number that uniquely identifies a particular network and enables Internet routers to forward a packet to its target network. The second component, known as the host ID (HostID) is the last number of the IP address. It is a number assigned to a particular computer (host) on

[1]Packets and Frames are the names given to Protocol data units (PDUs) at different network layers

[2]It is an unique address for each node, e.g. network interface card at the time of its manufacture

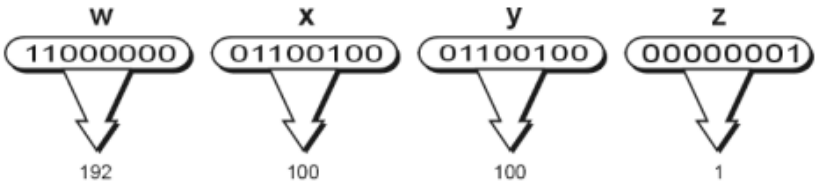

Figure 4.2: IP address structure

the destination network that enables the router serving that host to deliver the packet to the host directly. The computer must know which digits of the device address must match those of the network address. Doing so it is possible to separate the HostID from the NetID values and it is done by defining the so called subnet mask. The latter consists of four numbers that must take the value 0 or 255 depending on the class of the IP address. To set the NetID as the first three numbers it is necessary to set the subnet mask to 255.255.255.0. For instance, in the previous example 192.100.100.0 would be the NetID, and the computer or HostID would be 1. In the case study, a Wi-Fi router was used to route the messagges. The IP address were **169.254.123.5** for the UR5 and **169.254.123.1** for the PC. In order to set the PC IP address to **169.254.123.1**, it was connected to the network of the Wi-Fi router and the IPv4 address was set to static.

4.2.3 Transmission Control Protocol (TCP)

TCP is a connection-oriented protocol. Before sending data, TCP creates a connection between two hosts. Thanks to the established connection, the protocol is able to verify if a packet has been received and to arrange for re-transmission, if the packet is lost. TCP incurs a large extra expense in terms of processing time and header size because of all these built-in features. The features provided by the TCP are:

- Big data blocks divided into more manageable IP-compatible parts.
- Reconstruction of the data stream from the received packets.
- Communication of receipt.
- Socket services for multiple port connections on distant hosts.
- Packet checking and error correction

- Flow regulation.
- Sequencing and reordering of packets.

4.2.4 Ports

TCP must be aware of the process, i.e., software program, on that specific system the message is intended for, unlike IP, which may route the message to the machine based on its IP address. Ports with a range of 1 to 65535 are used for this.

Sockets

The IP address (location) and port number (process) are merged into a functional address called a socket in order to specify both the location and application to which a certain packet is to be transmitted. The IP address and port number are both present in the IP header and the TCP header, respectively. Any data transfer with TCP requires the existence of a socket at both the source and the destination. Multiple sockets can be created via TCP and connected to the same port.

4.3 Robot and PC Connection

As written above, the chosen type of communication between robot and PC was the socket. The available ports of the robot, to use this type of communication, are reported in the following table.

Table 4.1: Port characteristics of the UR5

	Primary	Secondary	Real-Time	Real-Time Exchange (RTDE)
Port no.	30001/30011	30002/30012	30003/30012	30004
Frequncy [Hz]	10	10	125	125
Receive	URScript commands	URScript commands	URScript commands	Various data
Transmit	Various data	Various data	Various data	1 integer 139 doubles 1116 bytes in total

To set up the connection a python code was written. This code uses a python library called **socket**. Built-in functions of this library make it possible to create a socket endpoint on both PC and the robot.
First, the IP address of the UR5 is set as HOST and that of PC as the server. Then, a port is chosen for data exchange. For the purpose of this work, port 30003 was chosen. The port works at a frequency of 125 Hz and can receive URScript commands.

4.4 Data Exchange Overview

It is necessary to make an overview of how the data are exchanged between the PC, the Realsense and the Robot. Figure 4.3 is a scheme of how data are exchanged between devices.

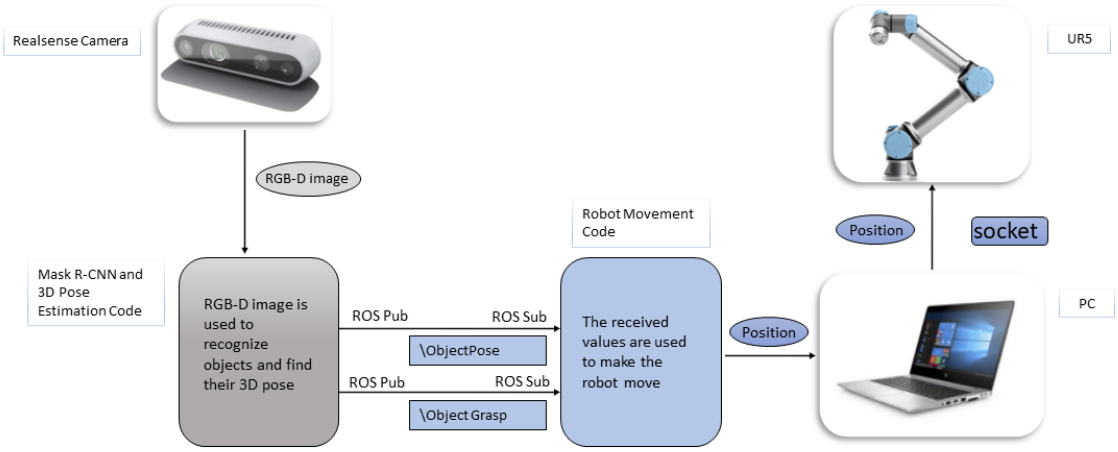

Figure 4.3: Data Exchange Scheme

As can be seen from the image, the RGB-D image is taken by Mask R-CNN and Pose Estimation Code. Once the code has performed the computations,

the results are published as ROS topics, using two ROS publishers. These topics are received by two ROS subscribers created in the code that moves the robot. Once these topics are elaborated, PC sends the robot a position to move to. Two socket endpoints are set up between the PC and the robot.

It is also worth to analyze how each code communicates with its classes. Figure 4.4 shows how the classes of the Mask R-CNN and 3D pose estimation code communicates. This code is divided into three different sub-codes. Two of them are the classes Realsense and Mask R-CNN. The last one, instead, called `measure_object_dinstance_pose_ros`, is the main code.

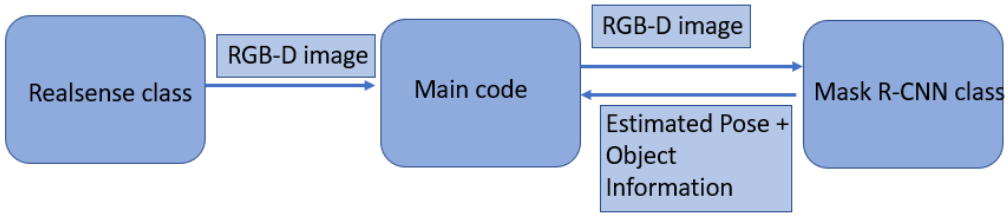

Figure 4.4: Mask R-CNN classes communication

In the main code, two objects are created, one of the Realsense class and one of the Mask R-CNN class Using the Realsense object, the main code starts camera streaming and provides the RGB-D images as input to the Mask R-CNN object. The latter object returns to the main code the estimated pose and some object information, such as the dimensions of the bounding box and its class.
Concerning the Robot movement code its internal communication structure is shown in figure 4.5. This code is divided into a class called PYUR and a main code.

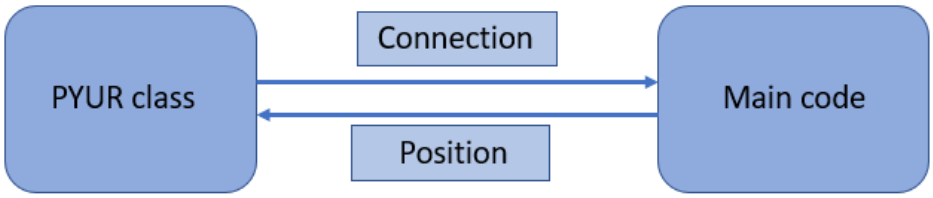

Figure 4.5: Robot Movement class communication

In the main code a PYUR object is created. The figure shows that the PYUR class receives from the main code the position where the robot has to be moved. The main code, on the other hand, receives the connection information as input. More details about this code will be available in the following paragraphs.

4.5 PYUR Class

To establish the connection between the robot and PC and to manage their communication, a Python class, called PYUR, was created. This class has two different methods. One is responsible for creating the socket endpoints on the robot side and on the PC side. The second method is used to send commands to the robot via PC.
The following Python code was written to implement the connection between the two devices.

```
def initialization():
    HOST= '169.254.123.5'
    server='169.254.123.1'
    PORT=30003
    # TCP/IP connection
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((HOST, PORT))
    return(HOST,server,s)
```

The first method of the class PYUR is called `initialization()`. The command `s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)` creates a socket object `s`. `socket.socket()` receives two different inputs. The first

is the socket_family, which defines the type of address needed. For the purpose of this work, the IPv4 address was chosen. Therefore, to comply with this statement `socket.AF_INET` is written. The second input is the socket_type, which instead specifies the type of socket. In this case, since a TCP socket type was chosen `SOCK_STREAM` is given as input.
The next command `s.connect((HOST,PORT))` allows to connect to the selected host through the previously defined port. Thanks to this simple procedure the robot and the PC can now send data to each other.

The second method of the class PYUR is `movel(s,pos)`. This method was designed to check if the robot can move in a given position, and then move it.

```
def movel(s,pos):
    if pos[0] < -0.8  or pos[0] >  0.8:
        print("Out of range in the X-axis!!")
        s.close()
        quit()
    elif pos[1] < -0.8 or pos[1] > 0.8:
        print("Out of range in the Y-axis!!")
        s.close()
        quit()
    elif pos[2] < 0.01 or pos[2] > 1:
        print("Out of range in the Z-axis!!")
        s.close()
        quit()
    else:
     print("The robot is Moving! ")
     time.sleep(2)
     posa = ', '.join(map(lambda x: f"{x:.3f}", pos))
     movel = f'movel(p[{posa}], 0.3, 0.1)\n'
     s.send(movel.encode()) # Send movel to URSim
     print( "URScript command movel: ") # Print information
     print( "{}".format(movel))
    return()
```

The `movel(s,pos)` methods has as inputs a socket object (`s`) and a point in the robot workspace (`pos`). First, a check of the specified X-Y-Z coordinates is performed. The code checks whether the `pos` values are within the robot's permitted range of motion. If the point is within the specified range, a command is sent to the robot, using a method of the

socket class `s.send()`. The sent message is the string representation of the `movel()` command. `movel()` is part of the URScript instruction that the robot control unit can receive. This command makes the end effector of the robot move linearly between waypoints. The intended end effector speed and acceleration, expressed in mm/s and mm/s^2, are the variables that may be configured for this movement type.

4.6 UR5 Grasping

In this work, two distinct codes were written concerning the movement of the robot. The first `Command UR Vision` receives the 3D position of the object and makes the robot to move according to the passages already explained in 3.5. The second code named `Command UR Vision Follow` was written with the structure of the first code, but it can also check whether the object to be grasped has moved during the grasping process. If this is the case, it can determine the new position of the object and make the robot move there.

4.6.1 Command UR Vision

```
1 [HOST, server, s] = PYUR.initialization()
2 rospy.init_node('PoseNode', anonymous=True)
```

First, the method `initialization()` of the class PYUR is called. As already explained, it is used to establish the connection between the robot and the PC. In the second line, a ROS node (`'PoseNode'`) is initialised, which is used to obtain the required information from the 3D position estimation code.

```
1  home_pos = [0.219,0.127, 0.416, 2.247, -2.247,0.059]
2 PYUR.movel(s, home_pos)
3 time.sleep(3)
```

After the initialisation phase, the robot is ready to be moved. As said in section 3.5, the grasping operation is divided into 4 steps. The first step is to send the robot arm to the so-called *Home position*. This is done by writing the `home_pos` waypoint and giving it to the PYUR method `PYUR.movel(s,home_pos)`. Figure 4.6 shows the robot in the Home position.

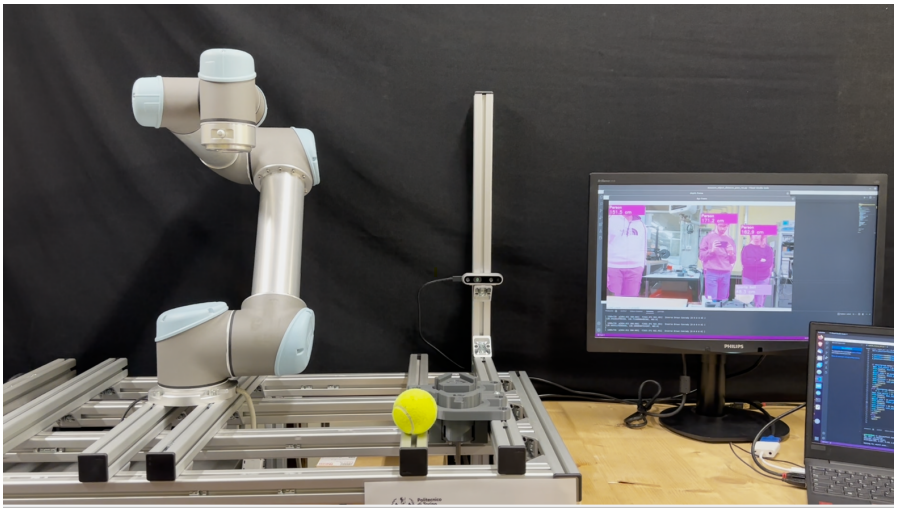

Figure 4.6: Home Position

```
 #Camera Pose
print("Waiting for object pose...")
pose_sub=rospy.wait_for_message("/ObjectPose",Pose)
objectgrasp=rospy.wait_for_message("/ObjectGrasp",String)
print("Object pose found!")
#Here we have to do frame transformation
Rotation_matrix=np.array([[0,0,1],[-1,0,0],[0,-1,0]])
#Converting the position to a suitable value for the simultion
Camera_x=round(pose_sub.position.x/1000,3)
Camera_y=round(pose_sub.position.y/1000,3)
Camera_z=round(pose_sub.position.z/1000,3)
#Dinstance between the world and the camera frame
Frame_dinst=np.array([-0.26,0.53,0.165])
#Point in the camera frame
Point=np.array([Camera_x,Camera_y,Camera_z])
#Convertion from the camera to the worls frame
World_position=Rotation_matrix.dot(Point)+Frame_dinst
X_position=round(World_position[0],2)
Y_position=round(World_position[1],2)
Z_position=round(World_position[2],2)
```

The part of code above defines two different ROS subscribers. One receives the estimated coordinate of the found object from the pose estimation code, while the second receives a topic which defines the orientation of the robot

end effector. The latter topic `objectgrasp` is a string, which can have two possible values *Top* or *Side*. This topic comes from the pose estimation code, too. During the object recognition a bounding box around the found object is drawn. The dimensions of the bounding box are used as a distinguishing feature to assign the value of `objectgrasp`. Its value depends on the height and width dimensions. Depending on which dimension is the largest, one of the two values of `objectgrasp` is assigned. If the height dimension of the bounding box is greater than the width dimension, the subscriber receives the value `Side`. If the height dimension is less than or equal to the width dimension, it will receive the value `Top`. This distinction was made to select the grasping position for the object. Therefore, the code is able to distinguish whether an object has to be grasped on the side or from the top.
The rest of the code transforms the received X-Y-Z position from the camera frame to the base frame of the UR5 robot (Transformation 3.1).
Figure 4.7 shows the UR5 base reference frame.

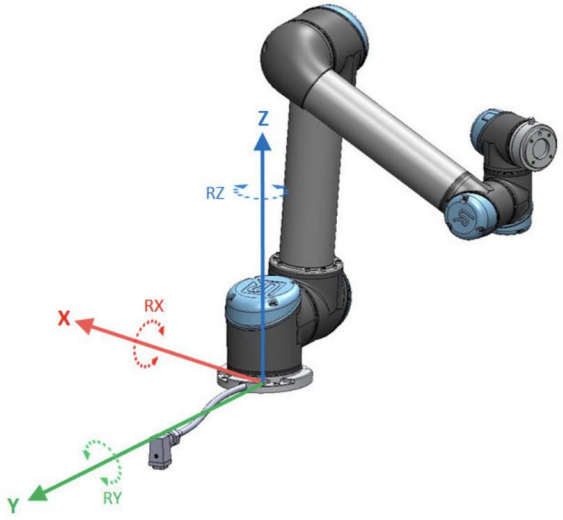

Figure 4.7: UR5 Base Reference Frame

```
1 if objectgrasp.data=="Top":
2 Near_found_object = [X_position, Y_position,Z_position+0.3,
    2.247, -2.247,0.059] #Position close to the target + Z
    offset
3 PYUR.movel(s, Near_foud_object)
4 time.sleep(3)
```

```
5
6  elif objectgrasp.data=="Side":
7  Near_found_object = [X_position, Y_position-0.2,Z_position+0.2,
       0.063,-2.181,-2.245] #Position close to the target + Z
     offset, -Y offeset
8  PYUR.movel(s, Near_foud_object)
9  time.sleep(5)
```

Once the conversion is performed, the robot can be moved to a position close to the object. Depending on the topic saved inside `objectgrasp` a different value for the `Near_found_object` waypoint is set. If the `objectgrasp` string is equal to `Top` the robot moves near the estimated object position with an offset on the Z-axis and the end effector facing down. Instead, if the `objectgrasp` string value is `Side` the robot moves close to the object position with, an offset, both on the Y-axis and the Z-axis. In the `Side` case the end effector is approaching the object from one side. Figure 4.8 shows the robot in the Close position with the `objectgrasp` value set to `Top`.

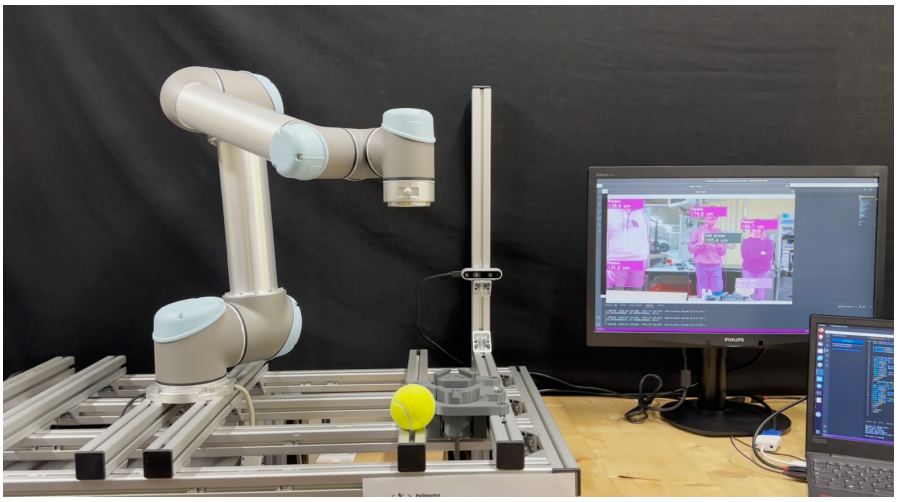

Figure 4.8: Approaching Position with `objectgrasp` set to `Top`

```
1  Object_pose = [X_position, Y_position,Z_position+0.1, 2.247,
     -2.247,0.059] #Grasping Position
2  PYUR.movel(s, Object_pose)
```

```
Object_pose = [X_position, Y_position-0.1,Z_position,
  0.063,-2.181,-2.245] #Grasping Position
PYUR.movel(s, Object_pose)
```

In the next phase, the robotic arm is moved into the grasping position. In both of the cases reported above, the grasping position corresponds to the object position, with a smaller offset, compared to the previous one. Going into details the first code puts a smaller offset on the Z position, instead, the second imposes an offset only on the Y-axis. Figure 4.9 shows the robot in the Grasping position.

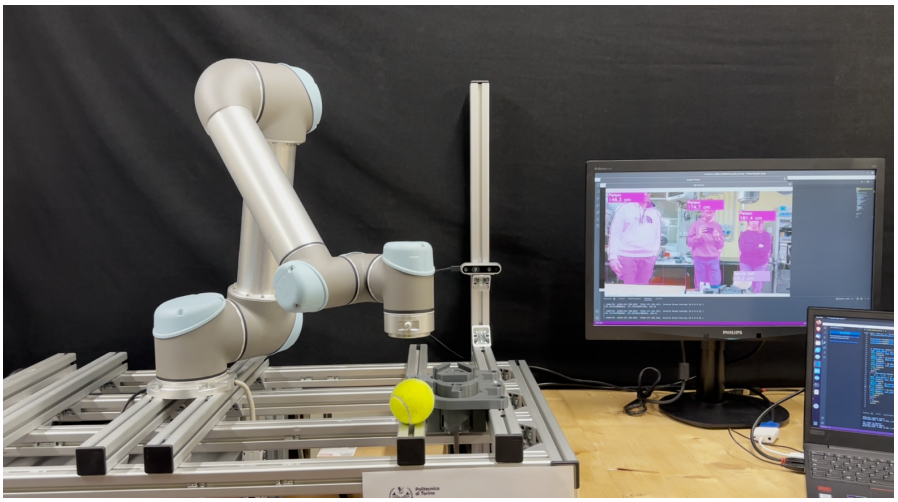

Figure 4.9: Grasp Position

```
Last_position = [0.219,0.127, 0.416, 2.247, -2.247,0.059] #
  Last position
PYUR.movel(s, Last_position)
```

At last, the robot is moved to a final position. In this case the final position corresponds to the home position. Figure 4.10 shows the robot in the Last position.

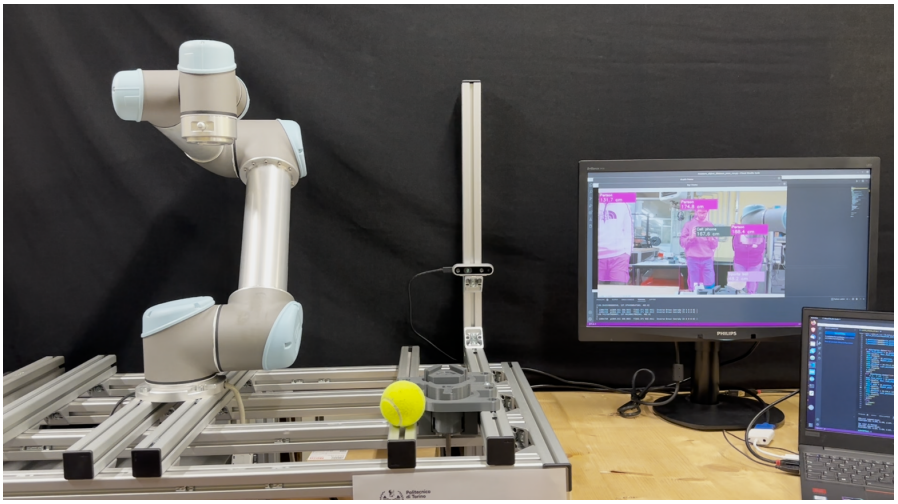

Figure 4.10: Final Position

```
s.close()
quit()
```

The two final lines of code are used to close the socket communication and make the coding process end.

4.6.2 Command UR Vision Follow

The previous code was modified in order to make the robot understand if the object has been moved, during the grasping procedure, or not. Doing so the robotic arm can adjust its position and move toward the object.

```
1   def check_position(X,Y,Z,X_old,Y_old,Z_old):
2
3  if X!=X_old or Y!=Y_old or Z!=Z_old:
4      move=True
5  else:
6      move=False
7  return move
```

The above function `check_position(X, Y, Z, X_old, Y_old, Z_old)` checks if the current position and the old one are equal or not. If it founds that, the two positions are different , it sets a flag value to `True`, whereas, if the two are equal it sets the flag to `False`. The `move` flag is used to understand if the robot has to move one more time or can go on with the grasping procedure.

```
1
2   if objectgrasp.data=="Top" and move:
3    Near_foud_object = [X, Y,Z+0.3, 2.247, -2.247,0.059] #
     Position close to the target + Z offset
4    PYUR.movel(s, Near_foud_object)
5    time.sleep(3)
6    wx=2.247
7    wy=-2.247
8    wz=0.059
9    X_old=X
10   Y_old=Y
11   Z_old=Z
12   Y_off=0
```

```
elif objectgrasp.data=="Side" and move:

 Near_foud_object = [X, Y-0.2,Z+0.2, 0.063,-2.181,-2.245] #
 Position close to the target + something Z
 PYUR.movel(s, Near_foud_object)
 time.sleep(3)
 wx=0.063
 wy=-2.181
 wz=-2.245
 X_old=X
 Y_old=Y
 Z_old=Z
 Y_off=-0.1
```

The two codes reported above are very similar to those described in the previous section. The difference is the check on the `move` flag. If `move` is true, the robot has to adjust its position and move one more time toward the object.

```
if not(move):
 Object_pose = [X, Y+Y_off,Z+0.1, wx,wy,wz] #Position close to
  the target
 PYUR.movel(s, Object_pose)
 time.sleep(3)
 exit=False
```

In case the move flag is set to False, the robot can go into the grasping position.
During the code test, the robot was able to follow the object. However, since no graphics processor was available at this stage, the robot cannot react quickly when the object changes position. This is because each time the object is moved, the 3D position estimation must be performed again, which can take some time

Chapter 5

Conclusion and Future Development

The goal of this work is to detect an object from an RGB-D camera image and grasp it with a robotic arm. The camera used to capture the RGB-D image is an Intel® RealSense™ D435. The RGB image is the input of a neural network that can detect an object and draw a bounding box around it. The network used is a Mask R-CNN specialized in detecting objects and drawing bounding boxes and masks around the detected object. An already trained network is employed so that only a certain number of objects can be detected. The depth image is used, together with the image center and focal length, to perform an inverse transformation and calculate the 3D coordinates of the detected object. To start camera streaming and convert these images into the required input for the mask-R-CNN, a Python code is used. Using the library `pyrealsense2`, it is possible to enable both RGB and depth streaming. Once the image has been processed, it is converted into `NumPy arrays`, which is the proper input for the neural network.
When the R-CNN mask detects an object, a bounding box is drawn around it. At this point, the 3D coordinate of this object can be retrieved doing an inverse projection transformation.
This measurement is performed 3 times, to get a better estimate of the 3D position. Afterwards, the calculated object coordinates can be passed to a ROS publisher. The published topic is received by a ROS subscriber, which is part of the Python code written for the robot control.

The robot employed for grasping is the UR5, CB3 series. This robot is at first simulated in the virtual environment GazeboSim, where it is controlled using a set of ROS packages and tools called MoveIt!.
Then, the real UR5 is deployed. The robot is controlled with a Python code. With this code, it is possible at first to set up two socket endpoints between PC and the robot, using the socket library, and then control the robot using the `.send()` command. This allows to control the robot by entering the URScript command `movel()` as input. `movel()` is a simple command that makes the robot end effector move linearly between waypoints. In this case, the waypoints corresponded to the movement the robot must make to reach the grasping position. The end effector of the robot can approach the object either from one side or from above, depending on the dimensions of the drawn bounding box. The distinction is given by the largest size between the height and the width of the bounding box.

Future developments of this project could be:

1. The use a PC with a powerful GPU, to speed up the process.

2. Train the Neural Network with a specific set of objects. These objects may be specific objects that the robot must grasp. In this work, only a tennis ball and a bottle were tested. This training could certainly improve the object recognition part.

3. Use of a different Network. It might be possible to use a different Neural Network, perhaps one of those mentioned in the second chapter. Switching to a Network that does not recognize the objects, but finds a grasping point directly could improve the overall speed of the system and also the success rate in grasping an object.

Bibliography

[1] Fabian Hutmacher. «Why Is There So Much More Research on Vision Than on Any Other Sensory Modality?» In: *Frontiers in Psychology* 10 (2019). ISSN: 1664-1078. DOI: 10.3389/fpsyg.2019.02246. URL: https://www.frontiersin.org/articles/10.3389/fpsyg.2019.02246 (cit. on p. 1).

[2] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing: Analysis and Machine Vision*. 2nd ed. CL-Engineering, 1998. ISBN: 053495393X (cit. on pp. 1, 2).

[3] Ken Goldberg. «MIT Robotics - The New Wave in Robot Grasping». In: Dec. 2019. URL: https://www.youtube.com/watch?v=ATDrSWZXuwk&t=770s (cit. on p. 8).

[4] D.G. Lowe. «Object recognition from local scale-invariant features». In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150–1157 vol.2 (cit. on p. 9).

[5] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. «SURF: Speeded Up Robust Features». In: *Computer Vision – ECCV 2006*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417. ISBN: 978-3-540-33833-8 (cit. on p. 9).

[6] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. «BRIEF: Binary Robust Independent Elementary Features». In: *Computer Vision – ECCV 2010*. Ed. by Kostas Daniilidis, Petros Maragos, and Nikos Paragios. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 778–792. ISBN: 978-3-642-15561-1 (cit. on p. 9).

[7] Federico Tombari, Samuele Salti, and Luigi Di Stefano. «Unique Signatures of Histograms for Local Surface Description». In: *Computer Vision – ECCV 2010*. Ed. by Kostas Daniilidis, Petros Maragos, and Nikos Paragios. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 356–369. ISBN: 978-3-642-15558-1 (cit. on p. 10).

[8] Yulan Guo, Mohammed Bennamoun, Ferdous A. Sohel, Jianwei Wan, and Min Lu. «3D free form object recognition using rotational projection statistics». In: *2013 IEEE Workshop on Applications of Computer Vision (WACV)*. 2013, pp. 1–8. DOI: 10.1109/WACV.2013.6474992 (cit. on p. 10).

[9] Ajmal S. Mian, Bennamoun, and Robyn A. Owens. «Three-Dimensional Model-Based Object Recognition and Segmentation in Cluttered Scenes». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28 (2006), pp. 1584–1601 (cit. on p. 10).

[10] Min Lin and SY Qiang Chen. «Imagenet classification with deep convolutional neural networks». In: (2012) (cit. on pp. 11, 24).

[11] Nov. 2022. URL: https://www.intelrealsense.com/depth-camera-d435/ (cit. on p. 12).

[12] June 2022. URL: https://pysource.com/2021/06/24/identify-and-measure-precisely-objects-distance-with-deep-learning-and-intel-realsense/ (cit. on pp. 15, 28, 36).

[13] June 2022. URL: https://pointclouds.org/ (cit. on p. 20).

[14] Radu Bogdan Rusu and Steve Cousins. «3D is here: Point Cloud Library (PCL)». In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 1–4. DOI: 10.1109/ICRA.2011.5980567 (cit. on p. 20).

[15] Aitor Aldoma, Zoltan-Csaba Marton, Federico Tombari, Walter Wohlkinger, Christian Potthast, Bernhard Zeisl, Radu Bogdan Rusu, Suat Gedikli, and Markus Vincze. «Tutorial: Point Cloud Library: Three-Dimensional Object Recognition and 6 DOF Pose Estimation». In: *IEEE Robotics Automation Magazine* 19.3 (2012), pp. 80–91. DOI: 10.1109/MRA.2012.2206675 (cit. on p. 20).

[16] Andreas ten Pas, Marcus Gualtieri, Kate Saenko, and Robert Platt Jr. «Grasp Pose Detection in Point Clouds». In: *CoRR* abs/1706.09911 (2017). arXiv: 1706.09911. URL: http://arxiv.org/abs/1706.09911 (cit. on p. 21).

[17] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojea, and Ken Goldberg. «Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics». In: *CoRR* abs/1703.09312 (2017). arXiv: 1703.09312. URL: http://arxiv.org/abs/1703.09312 (cit. on p. 21).

[18] Sulabh Kumra, Shirin Joshi, and Ferat Sahin. «Antipodal Robotic Grasping using Generative Residual Convolutional Neural Network». In: *CoRR* abs/1909.04810 (2019). arXiv: 1909.04810. URL: http://arxiv.org/abs/1909.04810 (cit. on p. 22).

[19] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, and Stan Birchfield. «Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects». In: *CoRR* abs/1809.10790 (2018). arXiv: 1809.10790. URL: http://arxiv.org/abs/1809.10790 (cit. on p. 23).

[20] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. «Mask R-CNN». In: *CoRR* abs/1703.06870 (2017). arXiv: 1703.06870. URL: http://arxiv.org/abs/1703.06870 (cit. on pp. 23, 27).

[21] June 2022. URL: https://cocodataset.org/#home (cit. on p. 24).

[22] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. «Rich feature hierarchies for accurate object detection and semantic segmentation». In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524 (cit. on p. 24).

[23] Ross B. Girshick. «Fast R-CNN». In: *CoRR* abs/1504.08083 (2015). arXiv: 1504.08083. URL: http://arxiv.org/abs/1504.08083 (cit. on p. 25).

[24] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. «Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks». In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497 (cit. on p. 26).

[25] Nov. 2022. URL: https://towardsdatascience.com/inverse-projection-transformation-c866ccedef1c (cit. on p. 32).

[26] Lentin Joseph and Jonathan Cacace. *Mastering ROS for Robotics Programming - Second Edition: Design, Build, and Simulate Complex Robots Using the Robot Operating System*. 2nd. Packt Publishing, 2018. ISBN: 1788478959 (cit. on pp. 45–47, 53, 55).

[27] Nov. 2022. URL: https://medium.com/swlh/part-3-create-your-first-ros-publisher-and-subscriber-nodes-2e833dea7598 (cit. on p. 48).

[28] June 2022. URL: https://wiredworkers.io/universal-robots-ur5/ (cit. on p. 48).

[29] Universal Robot. *UR5 User Manual*. English (cit. on p. 49).

[30] Nov. 2022. URL: https://www.universal-robots.com/articles/ur/interface-communication/overview-of-client-interfaces/ (cit. on p. 51).

[31] June 2022. URL: https://roboticscasual.com/ros-tutorial-how-to-create-a-moveit-config-for-the-ur5-and-a-gripper/ (cit. on p. 55).

[32] Nov. 2022. URL: https://classic.gazebosim.org/tutorials?cat=guided_b&tut=guided_b1 (cit. on p. 63).

[33] Sept. 2022. URL: https://www.theconstructsim.com/ (cit. on p. 64).

[34] Deon Reynders and Edwin Wright. «7 - Host-to-host (transport) layer protocols». In: *Practical TCP/IP and Ethernet Networking for Industry*. Ed. by Deon Reynders and Edwin Wright. Oxford: Newnes, 2003, pp. 122–132. ISBN: 978-0-7506-5806-5. DOI: https://doi.org/10.1016/B978-075065806-5/50007-7. URL: https://www.sciencedirect.com/science/article/pii/B9780750658065500077 (cit. on p. 72).

[35] Deon Reynders and Edwin Wright. «5 - Introduction to TCP/IP». In: *Practical TCP/IP and Ethernet Networking for Industry*. Ed. by Deon Reynders and Edwin Wright. Oxford: Newnes, 2003, pp. 74–77. ISBN: 978-0-7506-5806-5. DOI: https://doi.org/10.1016/B978-075065806-5/50005-3. URL: https://www.sciencedirect.com/science/article/pii/B9780750658065500053 (cit. on p. 72).

[36] Deon Reynders and Edwin Wright. «6 - Internet layer protocols». In: *Practical TCP/IP and Ethernet Networking for Industry*. Ed. by Deon Reynders and Edwin Wright. Oxford: Newnes, 2003, pp. 78–121. ISBN: 978-0-7506-5806-5. DOI: https://doi.org/10.1016/B978-075065806-5/50006-5. URL: https://www.sciencedirect.com/science/article/pii/B9780750658065500065 (cit. on p. 72).