UNIVERSITY OF PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

Master degree in Control Systems Engineering

Developement of a Matlab Interface for a Vision-Guided Robotic Arm

Supervisor
Professor Giovanni Boschetti

Master Candidate
Valeria Bianca Fantini

Academic Year
2022–2023

”The strength of your will
Leading the rising dawn
With noble elegance
Sparkling on.”
-Bloody Stream

Abstract

This thesis contains the implementation of the Matlab interface used to communicate to a 6-axis manipulator. The TM5-700 is a cooperative robot on whose arm is mounted an Eye-in-Hand camera. This interface is then integrated into a framework where also an HTTP server is needed to retrieve the pictures taken from the robot camera.

This vision system is then used to detect a target object to be subsequently picked up by the robot. To do so the binocular vision system model is employed in order to retrieve the the depth information of the 3D position of the target object. The ORB algorithm is utilized to locate the target object from the two pictures.

Once the 3D target position is retrieved, through the Matlab interface is possible to produce a script external to the robot that gives the robot instructions for the pick up routine.

At the end the experimental results of the deployment of the framework are presented, together with an estimation of the depth perception error.

Contents

Listing of figures

Listing of tables

Chapter 1

Introduction

The importance of robots in the industry field is ever increasing. One of the reasons for this is their accuracy and durability[1] which can be useful in a very wide range of tasks: from simple pick and place to precision welding, assembling and much more. Moreover the same arm can be reused for different tasks simply by changing the tool or adding to it. This makes the possible fields of deployment of such systems very large and heterogeneous.

In recent years the necessity for cooperation between robots and humans emerged as well. However, the methods required to regulate interaction and collaboration between humans and robots have not been fully established yet. These issues are the subject of research in the fields of physical human-robot interaction and collaborative robotics[2]. To provide incentives in this regard the *sensor-based control* study field arises and flourishes.

Computer vision is another field that is evolving and growing and it finds employment as one of the possible means that can help humans and robot coexist in the same workspace.

In this work the attention has been shifted on this type of sensor. In particular an Eye-in-Hand camera is mounted on the robotic arm to provide vision to the system. With a vision system, image processing and understanding follow. Although it is a complex and computationally expensive field, the richness of this type of information is rather unique[2].

1.1 Visually guided pick up problem

This thesis' objective is to build a framework for the `Techman TM5-700` cooperative robot to work through a `Matlab` interface in order to expand the capabilities of the robot especially in terms of exploiting advanced visual algorithms. An `HTTP` server is needed to handle photos produced by the camera mounted on the manipulator. The server is needed to run the object recognition algorithm as well.

One of the strength point of this work is reusability: in fact by adapting the Matlab class, the framework can be easily converted to be used by another robotic system.

The application that is developed in this context revolves around connecting the robot to an external device and expanding the capabilities of the manipulator as much as possible. This is pursued by building an interface through which is possible to give built-in robot commands but in a script form and not in a block scheme form. The `TMFlow` provides only this latter option when this thesis is written.
For this reason the main focus of the experiment is actually build this framework composed by:

- A Matlab interface that translates robot commands and transmit them to the robot;
- An `HTTP` server to handle images retrieved from the camera while the robot executes the `TMFlow` block scheme program.
- A `TMFlow` program that is able to receive the commands from the Matlab interface and send pictures to the `HTTP` server.

In order to test the robot external interface a pick up and place routine has been implemented. Although in this routine the camera images are used for **object detection** purposes. In particular the problem to be solved is to detect an object in the received photos and to calculate its 3D position using a single camera.

To do this normally two cameras in fixed positions are needed. Subsequently the binocular model is used to retrieve the depth information. But as suggested from the article [1] only one camera can be used to retrieve the two images and successfully make the calculation to retrieve the 3D position.

To correctly retrieve the 3D position from two images it is needed the estimation of the depth information, normally unavailable from one single picture. With two photos is possible to apply the binocular vision model, combine it with some geometric consideration and obtain such information.

To retrieve correctly the detected object's pose and plan the robot movement necessary to pick the target successfully is the purpose of solving the visually guided pick up problem.

1.2 Thesis overview

This thesis is organized as follows:

In the *second chapter* It is presented how the transformation matrices were obtained in order to translate positions between the three main frames used (*World*, *Robot* and *Image* frames). It is also described the two main algorithms employed for object recognition and depth estimation.

In the *third chapter* it is described the setup process to have all the three main components correctly set. The main components are the `TMFlow` software, the `HTTP` server running on *Visual Studio Code* and the Matlab class *TM*. It is also given some fundamental guidelines to use the stereo calibration script and the Matlab class.

In the *fourth chapter* it is given a brief description of the manipulator and gripper involved in the experiment. Subsequently it is described how the three main components of the framework work together. It is presented a flowchart of the working framework as a mean of better understanding the context. It is also presented a description of the relevant parts of the code used in this experiment. Finally It is presented the execution and the results of the visually guided pick up routine. It is also shown the result of the depth estimation upon repeated trials.

Finally in the *Conclusions chapter* it is exposed a summary of the whole project and some ideas for future employment of this framework.

Chapter 2

Theoretical background

In this chapter it is presented how the transformation matrices between the three main frames are obtained. Subsequently it is given a description of the two main algorithms and the binocular model on which the visually guided pick up routine is based.

They are used to recognize the object and derive its 3D position in the world coordinates. In particular the ORB (Oriented FAST and Rotated BRIEF) is used to carry out the feature matching method. The depth estimation is made through a geometric consideration.

2.1 Frames used and their transformation matrices

The adopted experimental setup is composed of the following systems:

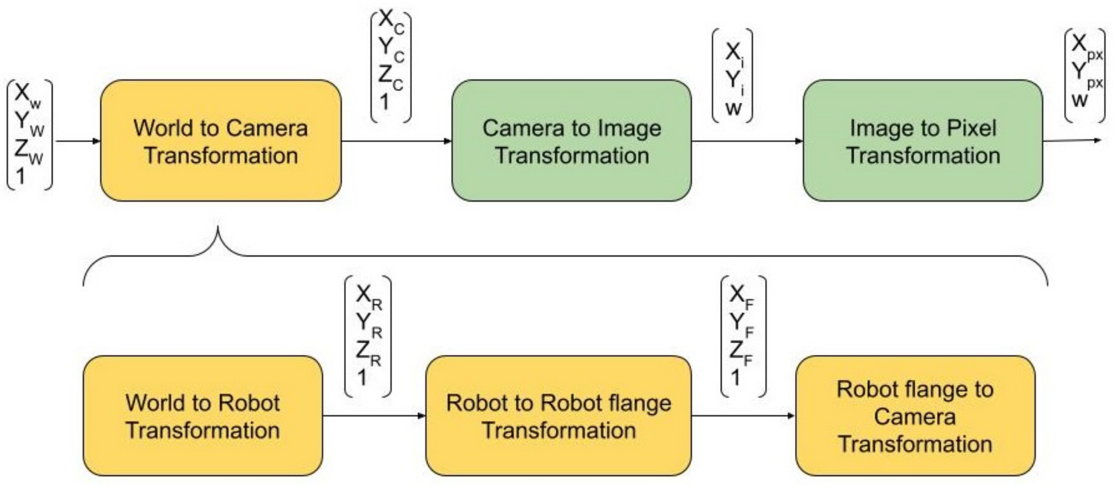
- The **table area** which is intercepted by the camera. Here it is placed the *World reference frame*;

- The **manipulator** which has its own *Robot reference frame*;

- The **camera** which is mounted on the robot arm. The camera concerns three different reference systems, two internal to the camera and one external to it. Hence an *Image reference system*, a *Pixel reference system* and a *Camera reference frame* are used.

The *World frame*, the *Robot frame* and the *Camera frame* are included into the **Extrinsic matrix** calculation. The *Image frame* and the *Pixel frame* are included into the **Intrinsic matrix** calculation. The composition of these two matrices gives the relation needed to translate *pixel* information into 3D *world* information, as shown in Fig.2.1.

In this experiment context the available information is the pixel position of the object detected through the `ORB` algorithm presented in 2.2. This means that in order to obtain the 3D position of the object, the product of the Intrinsic times the Extrinsic matrix must be inverted. The problem with this approach is that the depth information is lost in translation from 3D to 2D and so a separate evaluation of such value is needed.



$[X_W, Y_W, Z_W, 1]$
World to Camera Transformation
$[X_C, Y_C, Z_C, 1]$
Camera to Image Transformation
$[X_i, Y_i, w]$
Image to Pixel Transformation
$[X_{px}, Y_{px}, w]$
World to Robot Transformation
$[X_R, Y_R, Z_R, 1]$
Robot to Robot flange Transformation
$[X_F, Y_F, Z_F, 1]$
Robot flange to Camera Transformation

Figure 2.1: World to pixel transformation

2.1.1 Intrinsic matrix

The **Intrinsic matrix** is the transformation matrix that transforms 3D *camera* coordinates into 2D *pixel* coordinates. It is the composition of two separate transformations: the transformation from *Camera frame* to *Image frame* and the transformation from *Image frame* to *Pixel frame.*

The *Pixel reference frame* has the origin placed at the top left corner of the image with the positive x-axis pointing to the left and the y axis pointing downwards.
The 2D coordinate $[X_{px}, Y_{px}]$ in Fig.2.1 represents the *pixel* coordinate to be translated into *Image* coordinate. It is useful to work with homogeneous coordinates and hence in practice the 2D coordinate becomes $[X_{px}, Y_{px}, w]$ The transformation from *Image frame* to *Pixel frame* consists into a translation of the origin of the reference system from the center of the image to its top left corner. Hence the transformation matrix is of the form:

$$K_{IP} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

where the couple (f_x, f_y) is a scaling factor that considers going from a square pixel to a point. The resulting three dimensional homogeneous coordinate will be $[X_i, Y_i, w]$.

The *Image frame* is a coordinate system that has the 3D points in the camera coordinate system projected onto a 2D plane of a camera with a Pinhole Model. The 2D plane is what is captured as images by the camera. It is a lossy transformation, which means projecting the points from the camera coordinate system to the 2D plane can not be reversed.

The X_i and Y_i coordinates of the points are projected onto the 2D plane. The 2D plane is at

f (focal-length) distance away from the camera as shown in fig. 2.2. Hence the transformation matrix is obtained as:

$$K_{CI} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.2}$$

As said before the transformation is lossy and so inverting this transformation produces an incorrect evaluation of the third coordinate. However, this information in this context is retrieved from a separate calculation in section 2.3, instead the information about x and y is kept.

This time the input point is an actual 3D position but since it is added a dimension for homogeneous coordinate's sake the coordinate will be four dimensional and in the form $[X_c, Y_c, Z_p, 1]$ where Z_p is the distance between the camera lens and the target object (fig. 2.2).

All that has been proposed with regards the intrinsic matrix is true for a single camera. However when a two camera system is involved, the evaluation of the *World frame* coordinate on the object must be modified.

Referring to Fig.2.2 the following relationships are found applying the similarity theorem of the triangles[1]:

$$z_p = \frac{D}{\frac{|x_{i1}|}{f} + \frac{|x_{i2}|}{f}} \tag{2.3}$$

$$x_p = \frac{1}{2} z_p \left(\frac{x_{i1}}{f} - \frac{x_{i2}}{f}\right) \tag{2.4}$$

$$y_p = z_p \frac{y_{i1}}{f} = z_p \frac{y_{i2}}{f} \tag{2.5}$$

where D is the baseline and f the focal length found at 2.3.1.

It is very important to notice that z_p **is not** the result of the depth estimation exposed in 2.3, as such estimate is the sum of the focal length f and Z_p. Z_p is highlighted in Fig.2.2.

As a result the whole Intrinsic matrix is given by:

$$K_{intrinsic} = K_{CI} K_{IP} \tag{2.6}$$

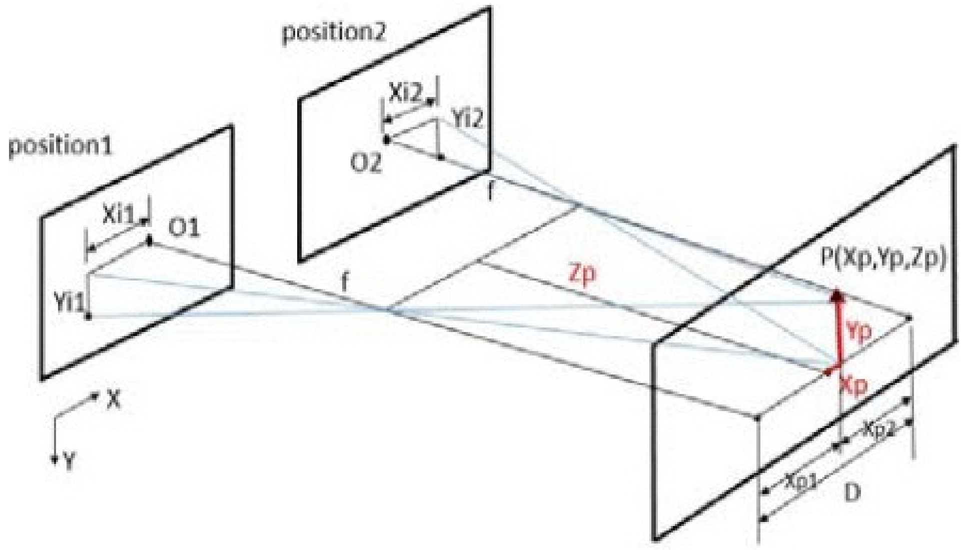


Figure 2.2: Binocular vision model. Picture from [1].

2.1.2 Extrinsic matrix

The **Extrinsic matrix** is the transformation matrix that transforms the 3D coordinates in the *World frame* into the 3D coordinate with respect to the *Camera frame*. In this case, as highlighted in Fig.2.1, the extrinsic matrix is the composition of two main transformation matrices: from *World frame* to *Robot frame* and from *Robot frame* to *Camera frame*.
For better picturing relative poses between frames figure 2.3 gives a schematic overview of the context.

The *World frame* has been positioned on the table area intercepted by the two cameras. In particular it has been measured a pre-determined 3D position through the `TMFlow` software and it is hence used to obtain the transformation matrix between *Robot frame* and *World frame*. The predetermined world frame origin will be noted as $[O_x, O_y, O_z, 1]_{Robot}$.

The *Robot frame* has its origin at the base of the robot and the z-axis pointing upwards, oriented as shown in fig. 2.3 with respect to the workspace area (in light blue). For convenience the *World frame* has the x and y axis parallel to the ones of the *Camera frame* and has the z axis pointing downwards. Hence the relative rotation between *Robot frame* and *World frame* is a rotation around the y axis of 180 degrees. The transformation matrix from *World frame* to *Robot frame* becomes:

$$T_{WR} = \begin{bmatrix} -1 & 0 & 0 & O_{x,Robot} \\ 0 & 1 & 0 & O_{y,Robot} \\ 0 & 0 & -1 & O_{z,Robot} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

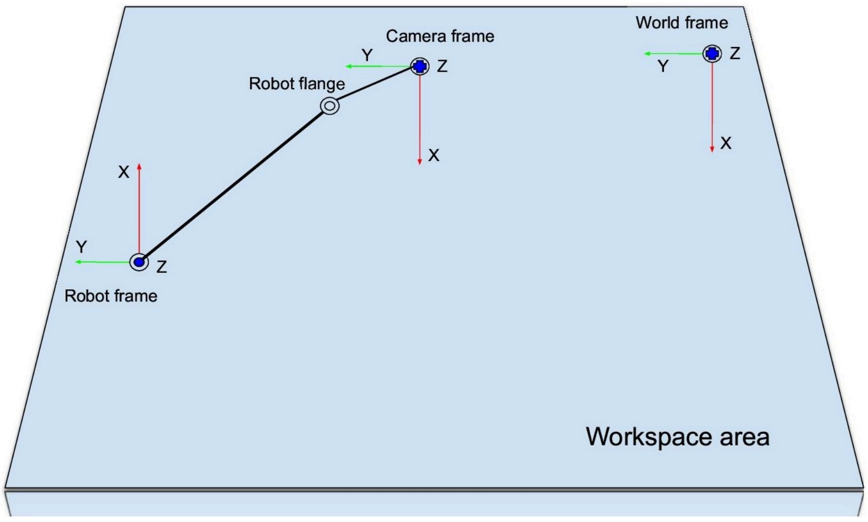


Figure 2.3: Schematic overview of relative frames poses

Since the camera is mounted on the last joint of the robot, in order to move the reference system from camera to the flange of the robot, it is important to remember about the physical distance between the two. Hence the transformation from the camera to Robot flange becomes:

$$T_{CaF} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 79 \\ 0 & 0 & 1 & 52.2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

where the explicit numbers are the physical offsets of the camera in millimeters with respect the robot flange. This transformation is a simple translation.

It is **very important** to notice that each set of coordinates returned from the *TMFlow* is the current position of the robot flange with respect to the *Robot frame* and this can cause a bit of confusion if not taken in consideration.

Hence the only frames used are the *World frame*, the *Robot frame* and the *Camera frame* which has the same orientation of the *World frame* but different origin position. The *Camera frame* is also related to the robot flange only by a rigid translation, simplifying in this way the calculations.

As a result the Extrinsic matrix appears as:

$$K_{extrinsic} = T_{CaF}T_{WR} \tag{2.9}$$

2.2 Object Recognition algorithm

The term `Object Recognition` refers to the set of vision tasks that involve identifying objects and their position from 2D images. We can distinguish three kinds of computer vision tasks related to object recognition:

- The **Image Classification** method that can predict the type or class of an object present in an image;
- The **Object Localization** method that can locate the presence of objects in a given image and indicate their position with a bounding box;
- The **Object Detection** method that combines the previous two methods and hence is able to predict the type of object present in the image and locate its position.

The specific algorithm used in this context is the `ORB` (Oriented FAST and Rotated BRIEF) and it is an Object Detection algorithm. Before describing `ORB`, it is presented the set of instruments vital to understand the mechanism behind this process.

2.2.1 Features, keypoints and descriptors

Features don't have a precise definition, they are rather specific patterns or specific attributes which are unique to an image and can be easily compared. A good example of what a feature is, it is given by the OpenCV documentation where the image in Fig.2.4 is presented:

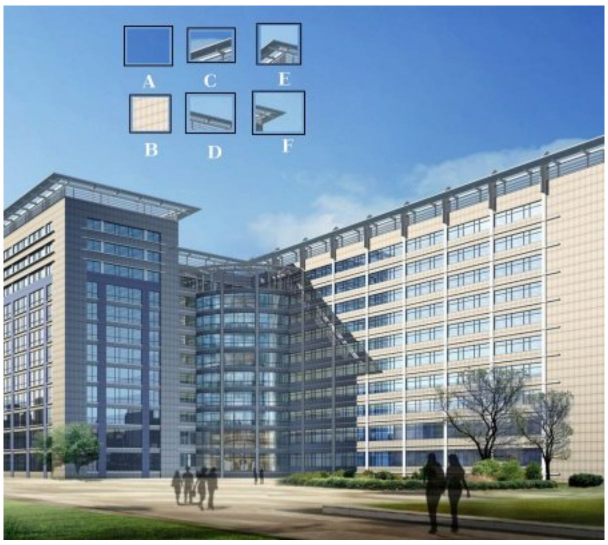


Figure 2.4: Image with possible recognizable features. Picture from [3]

If one tries to compare feature A (a piece of blue sky) with the image, it will be immediate to see that it is impossible to understand the exact position of where feature A is placed or how it is rotated with respect to the image. The only possible information is a very general location in the image (one can think about the sky being above everything). A similar reasoning goes for feature B.

It is natural now to try to find some more defined feature, for example an edge of the building (as shown in features C and D). Now the possible matching region is considerably reduced and it is also possible to have some information about the relative rotation of the feature with respect the image. It is however still not quite simple to identify the exact location of the edge of the building in either of the cases.

Finally one looks for a corner that is a strongly defined feature of the image and much more simple to identify inside the image in terms of either position and relative rotation. This is the fundamental intuition behind the searching of unique features of an image.

After having identified the unique features of an image (or anything that can be lead back to a corner) the next problem is trying to describe said feature in order to find other similar patterns in the same image. These described patterns are their pixel coordinates are the **keypoints** of an image. This description can be done in a number of different ways, but what the ORB uses is a modified version of the *Harris corner detector*.

The *Harris corner detector* finds the difference in intensity for a displacement of (u, v) in all directions. This is expressed in the formula:

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2 \tag{2.10}$$

where $w(x, y)$ is the window function and $I(x, y)$ is the intensity function. The function 2.10 is then maximized and it is obtained:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \tag{2.11}$$

with:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_y I_x & I_y I_y \end{bmatrix} \tag{2.12}$$

where I_x and I_y are the image derivatives in directions x and y respectively. Finally a score is checked, it determines if a window can contain a corner or not and it is given by the following equation:

$$R = det(M) - k(trace(M))^2 \tag{2.13}$$

where:

1. $det(M) = \lambda_1 \lambda_2$;
2. $trace(M) = \lambda_1 + \lambda_2$;

3. λ_1 and λ_2 are the eigenvalues of M.

The magnitude of these eigenvalues decide whether a region is a corner, an edge or flat by the following rule:

1. $|R|$ small $\Rightarrow \lambda_1$ and λ_2 are small $\Rightarrow$ the region is flat;

2. $R < 0 \Rightarrow \lambda_1$ » λ_2 or viceversa $\Rightarrow$ the region is an edge;

3. R is large $\Rightarrow \lambda_1$ and λ_2 are large and $\lambda_1 \sim \lambda_2 \Rightarrow$ region is a corner.

The *Harris corner detector* is rotation invariant but not scale invariant, and to this problem, D.Lowe came up with a new algorithm which extracts keypoints and compute its descriptors. This algorithm is called SIFT but is only distantly related to ORB. It is hence described FAST and BRIEF algorithms that constitute the basics for the ORB algorithm.

2.2.2 ORB

The `ORB` algorithm is a fusion of two algorithms: `FAST` (Features from Accelerated Segment Test) and `BRIEF` (Binary Robust Independent Elementary Features).

The `FAST` algorithm was proposed by Edward Rosten and Tom Drummond in their paper ”Machine learning for high-speed corner detection” in 2006 (Later revised it in 2010). The feature detecting technique used can be summarized as follows:

1. Select pixel p in the input image. This pixel has to be identified as an interest point or not. Let its intensity be I_p;

2. Select appropriate threshold value t;

3. Consider a circle of 16 pixels around the pixel to be tested;

4. p is a corner if there exists a set of n contiguous pixels in the circle which are all brighter than $I_p + t$ or all darker than $I_p - t$;

5. **High-speed test**: This test examines only the four pixels above, below, to the right and to the left of p (First above and below are tested if they are too brighter or darker. If so, then checks to the right and to the left). If p is a corner, then at least three of these must all be brighter than $I_p + t$ or darker than $I_p t$. If neither of these is the case, then p cannot be a corner. The full segment test criterion can then be applied to the passed candidates by examining all pixels in the circle.

A **keypoint descriptor** is a summary of an nxn neighbourhood around the pixel tested p. In particular once the size of the neighborhood is chosen, it can be subdivided and for each subsection it is analyzed the orientation of the keypoint and mapped into an histogram.

How the orientation is analysed and in how many section is subdivided the neighborhood depends on the type of algorithm that is being used. In summary descriptors can carry orientation information or color intensity gradient information.

The `BRIEF` algorithm is a feature descriptor which doesn't provide any method to find features. Descriptors can be very high dimensional vectors with floating point numbers (i.e. SIFT, another feature matching algorithm, uses 128-dimensional vectors). A memory handling problem can arise from using these descriptors.

After compressing descriptor information into binary strings, `BRIEF` takes smoothed image patches and selects a set of n_d (x,y) location pairs. Afterwards pixel intensity comparisons between the selected pairs are executed which result in a binary number. Piling up all the results from the intensity comparisons a n_d-dimensional bitstring is obtained. Working with bit strings rather than high dimensional floating point vectors makes `BRIEF` a fast algorithm.

As previously anticipated, `ORB` is a fusion of `FAST` and `BIEF`. In particular it uses `FAST` to find the keypoints, it applies *Harris Corner measure* to find the top N points among the identified keypoints.
Although `FAST` does not compute orientation information, `ORB` computes the intensity weighted centroid of the patch with located corner at the center. The direction of the vector from this corner point to centroid gives the orientation. To improve the rotation invariance, moments are computed with x and y which should be in a circular region of radius r, where r is the size of the patch.

Subsequently `ORB` uses `BRIEF` descriptors which unfortunately perform poorly with respect to rotation. To compensate for this `ORB` rotates the `BRIEF` descriptors according to the orientation information obtained from the keypoints at the previous step.
For any feature set of n binary tests at location (x_i, y_i) it defines a $2xn$ matrix called S which contains the coordinates of these pixels. After `ORB` uses the orientation of the patch θ to obtain the rotation matrix of the selected patch. It applies the matrix on S to obtain S_θ, the matrix of the oriented patch locations.

The detailed code for the `ORB` used in this context is proposed in section A.2 where the algorithm is integrated into the `HTTP POST` function. However in Fig.2.5 it is shown an example of output.

Here the *training input* is given by the image of the joypad box alone, which constitutes the object to find in the other input image, called *scene*. The ORB algorithm firstly finds all the points of interest (the keypoints) in the training image, assigns them an orientation and draws them on the picture in the form of light blue circles. It does the same in the scene image.

Secondly `ORB` employs a matcher object which matches the keypoints in the training image with the scene keypoints. A utility secondary function is then used to draw the red lines shown in the picture.

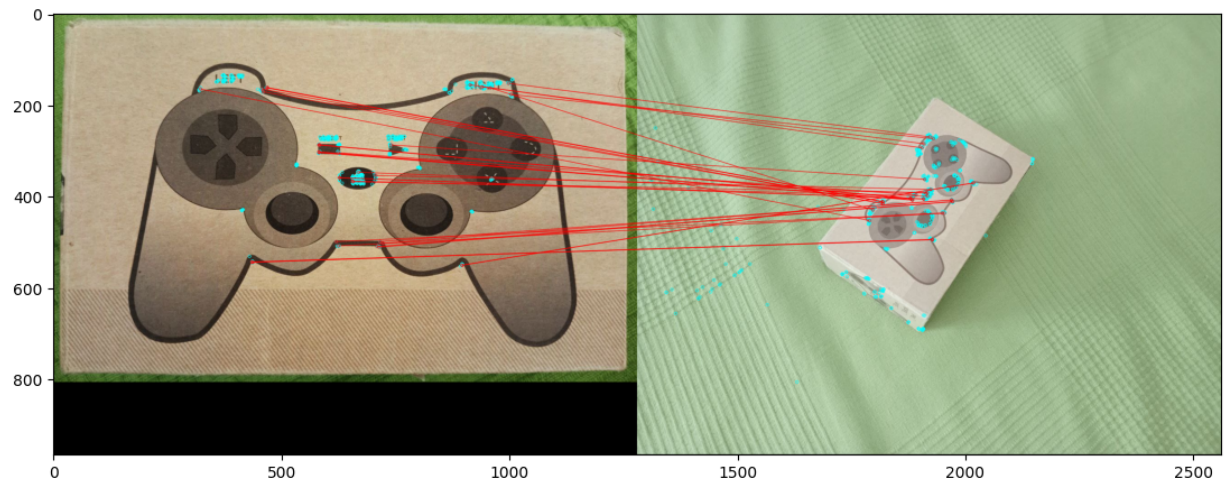


Figure 2.5: Application of the ORB algorithm to recognize a given object in an image.

2.3 Depth estimation

2.3.1 Focal length approximation

Focal length is one of the key instruments to estimate correctly the 3D position of the object in front of the camera. This information however is not present in the hardware manual of the robot. Luckily it is given the linear relationship between FOV and working distance of the camera, as shown in this table:

Working distance [mm]	FOV (width) [mm]	FOV (height) [mm]
100	96.9	72.7
300	281.6	211.2

Table 2.1: Linear relationship between camera FOV and working distance.

An initial working distance (`WD`) of 300mm has been chosen in order to have the view as wide as possible. The relationship used to evaluate the focal length comes from trigonometric considerations, for a better understanding the Fig. 2.6 illustrates the context.

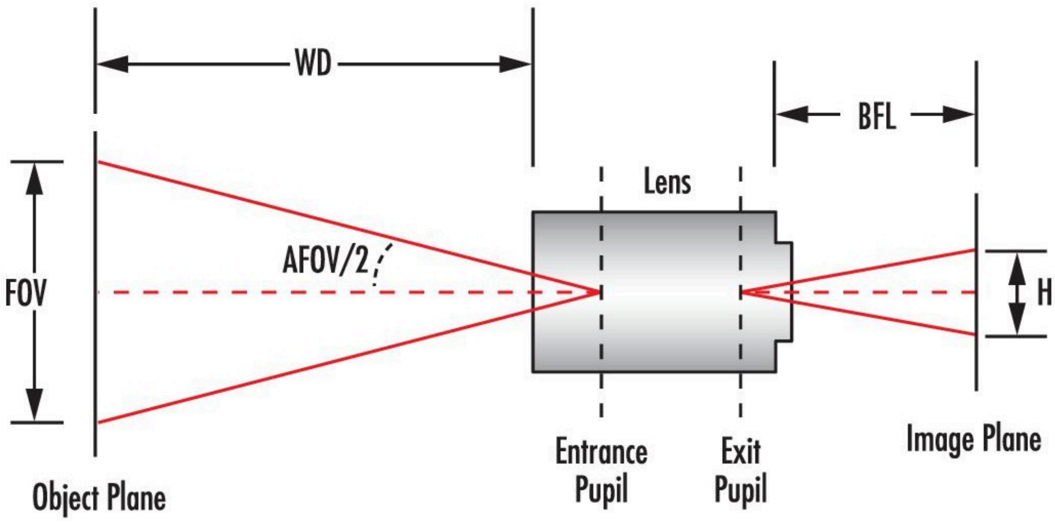


Figure 2.6: Representation of the relationship between focal length, FOV and working distance. Picture from [4].

As shown in Fig. 2.6 the relationship between the *Angular Field of View* $\frac{\text{AFOV}}{2}$ and $\frac{\text{FOV}}{2}$ is given by the Pitagora theorem

$$AFOV = 2 * tan^{-1}(\frac{FOV_{width}}{2 * WD}). \tag{2.14}$$

FOV_{width} is retrieved from table 2.1 considering $WD = 300mm$. Once the AFOV is obtained, the focal length can be approximated using:

$$F = \frac{H}{2 * tan(2 * AFOV)} \tag{2.15}$$

where H is the camera sensor width. The latter is unknown as well and it is retrieved using:

$$H = S_{px} * res \tag{2.16}$$

where S_{px} is the size of the pixel, *res* is the camera resolution in one direction (in this case in the width direction). S_{px} and *res* are evaluated using:

$$S_{px} = \frac{tan(\frac{FOV_{width}}{2}) * WD}{Image_{width}} \qquad Image_{width} = 2592[pixel]$$

$$res = \sqrt{(\frac{FOV_{width}}{FOV_{height}} * Mpx)} \qquad Mpx = 5M$$

FOV_{width} can be read from table 2.1. Mpx is the total camera resolution in mega pixel and res is the horizontal resolution of the camera. Finally the $Image_{width}$ is the width of the image in pixel. The resulting focal length is

$$F = 2.79mm. \tag{2.17}$$

2.3.2 Depth estimation

The depth evaluation is made by obtaining two distinct visuals of the object. In particular these two visuals are parallel with respect to each other as shown in Fig. 2.7. This context allows to simplify the evaluation since the y coordinate of the two cameras is the same. The only concern is toward the x and z coordinates of the object with respect to each *Image frame*.

In this experiment only one camera is available, mounted on the robot arm. Hence two photos need to be taken in two subsequent moments. The 3D positions with respect to the *World frame* of the two camera visuals are known thanks to the TMFlow software that allows to retrieve current poses of the robot, therefore also the baseline value is known. The focal length value is known at this point after the calculation made in 2.3.1.

The 3D point of the target object with respect to the *World frame* is projected onto the left and right image frames that is then translated into pixel coordinates which is actually the available information obtained through the `ORB` procedure described in 2.2.

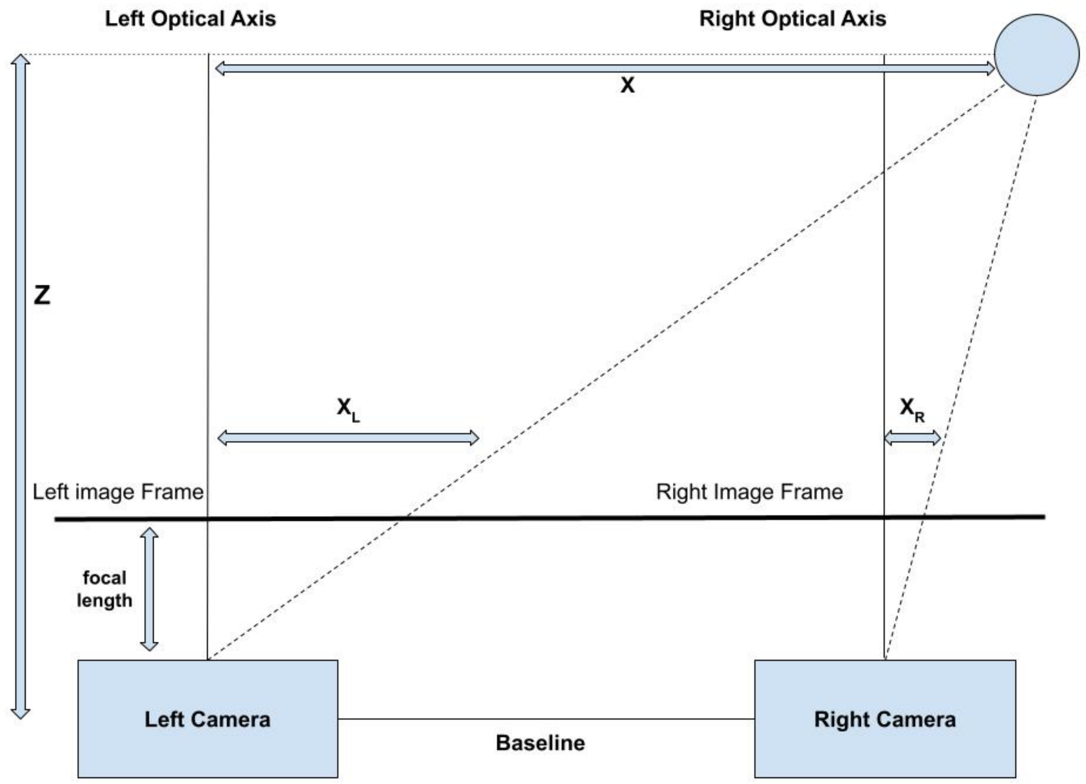


Figure 2.7: Multiview Geometry

The X_L and X_R shown in Fig.2.7 represent how much the object is distant from each respective optical axis. This information is used to evaluate the *Disparity* value which represents the difference in image location of the same 3D point from two different camera angles. This important value is used to evaluate the apparent motion in pixels for every point to produce intensity images and depth maps.

In this case X_L and X_R are equal to the x coordinate of the center of the box detected through the ORB procedure in the respective images.
From the stereo calibration step, described in section 3.2.1, the camera matrix of each camera is obtained and it is in the form:

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.18}$$

where the couple (f_x, f_y) is the focal length in pixel for the directions x and y of the pixel reference system. The couple (c_x, c_y) instead is the offset between the *pixel* reference system and the *image* reference system. This couple is different from the couple (c_x, c_y) that is evaluated from the ORB algorithm because that is the center of the box drawn around the detected object.

The depth estimation, Z in Fig.2.7, is obtained from:

$$Z = \frac{\|Baseline * f_x\|}{X_L - X_R}. \tag{2.19}$$

Chapter 3

Description of framework components and setup

This chapter is divided in three sections: one for each main component of this framework. It is described the setup for each part of the framework.

Firstly it is setup the `TMFlow` block scheme. Secondly the stereo calibration has to be done before starting the framework and storing the relevant information for later. Afterwards that the `HTTP` server is setup and finally the Matlab script is written. A similar pattern is followed in this chapter.

3.1 TMFlow setup

`TMFlow` is a graphical Human-Machine Interface (**HMI**). It uses blocks and line connections to describe the flow of commands to give to the robot. This software is integrated into the robot controller.
The logic of the program on the `TMFlow` is showed in 4.3. Since it is supposed to take two photos to satisfy the binocular model exposed in section 2.3, two separate vision jobs need to be set up in order to take and save the two photos. This is because it has been chosen to differentiate the two photos from the (key,value) pair of each POST request.

Hence the program starts with the two vision job nodes where the robot positions itself onto the pre-determined two 3D positions for the camera, takes the photos and sends them to the `HTTP` server. Each vision node expects a json file as an answer from the server containing:

1. Image size (depends on the TM vision image source, hence the camera).
2. Image format (jpg or png formats are available).
3. box_cx: center of the box, coordinate x in pixel.
4. box_cy: center of the box, coordinate y in pixel.

5. box_h: height of the box, in pixel.
6. bow_w: width of the box, in pixel.
7. label: name to give to the recognised object in the box.
8. rotation: rotation of the box with clockwise notation, in degrees.
9. score: score of accuracy of the recognition, a number between 0.00 and 1.00. In this project this evaluation is not implemented so it's always set to 1.
10. message: optional additional message.

Hence before passing each vision node the program waits for the json file answer. After having received the two answers the next step is connecting the robot to Matlab through the ***Network Node*** and the ***Listen Node***. Through the ***Listen Node*** the robot will receive the commands from the Matlab script.
The particular nodes which revolve around the whole project are: the ***Network Node***, the ***Listening Node*** and the ***Vision Node***.

3.1.1 Network Node

The ***Network Node*** is used to connect the robot to external devices. To work correctly it requires a previous setting in the **Network Settings** page of the `TMFlow` interface. But it can also be done directly from the node by choosing the **add device** option. In this page it is required a name for the external device, its **ip address** and its **port number**.

Now that the external device is setup, the ***Network Node*** can be used in the project.

To set up the node correctly it is required to enter the node and choose the previously setup external device. After this it is required to set the node to either *Receive from variable* or *Send* to decide between inbound or outbound traffic. In the first case messages are stored in a previously created and then selected variable. In the second case a message is sent to the `TMFlow` log window whenever this connection is established between robot and external device.

Next it is required to create and select a *Connection Status*. It is optional to set *Extra Idle Time* to be filled with either an existing variable or a time quantity in milliseconds.

3.1.2 Listen Node

The ***Listen Node*** is used to communicate with the external devices previously connected through the ***Network Node***. This node establishes a socket `TCPlistener` and has a specific message protocol whose explanation is tightly ed to the mechanism of the `TM` class in section 3.3 and hence will be focused on later.

The commands that will be passed to the ***Listen Node*** will be executed in order. If the command is not valid (for example a syntax error is present) an error message will be displayed in the log window on the `TMFlow` interface.

The commands are divided in two categories: instant commands and commands that need to be executed in sequence. To the first category commands like single movement commands or `InstantReadIO` (section 3.3.2) which are executed instantly. Instead for the second category the commands will be placed in a queue and will be executed in order `FIFO` as it happens for the `PVT` commands (section 3.3.2).

To set up correctly the Listen Node it is needed to set up the settings inside the Node. The connection timeout quantity refers to how much time in milliseconds the node waits to until a connection with the external device is established. Setting this number to zero means that the node waits indefinitely until the connection is established.

The Data Timeout quantity instead represents how much time in milliseconds the node waits for data from the connection. It is **very important** to set this value to a number different than zero, especially if the program to be executed is meant to loop continuously through the Listen Node.

In addition it is recommended to set a `WaitFor Node` on the fail path of the Node and make this path loop back to the Listen Node. In this way the node will continuously wait for new commands coming from the external device.

The Listen Node has two possible exit conditions: a *pass* condition and a *fail* condition.
The *pass* condition is fulfilled when the `ScriptExit()` command is executed 3.3.2.
The *fail* condition is fulfilled if one the following three events has happened:

- The connection timeout is elapsed;
- The data timeout is elapsed;
- The program flow has entered the current Listen Node before the TCP listener is started up.

3.1.3 Vision Node

The ***Vision Node*** is the principal node used to handle camera operations. This node provides the creation of a *Vision Base* that corresponds to the *Camera Frame* in the operations of this context. This node provides a variety of object recognition methods that work completely internally to the robot system and whose variables are nearly inaccessible or cryptic to an external device user.

However there is also a modality which allows *external detection* and *external classification*. The word external meaning that an external device can elaborate the camera images externally from the `TMFlow` and send back a json file containing relevant information. This is the key point that is exploited in this thesis' project.

Detection and **Classification** are two different **object recognition** approaches. Both these methods allow also photo sharing but only passing through an external `HTTP` server and hence it is explained the need of this ***vital*** component to this project.

To set up this node it is needed to open the node, navigate to the **Vision job** setting and here a window is opened. It is needed to navigate to `Task Designer` button to create the custom vision job.
In this case it has been selected AOI-only as the application for this vision task.

From here a flowchart of the task appears with the first block used to tune camera parameters for colour, light and focus adjustments. It is possible to add more blocks to enhance the quality of the image. More on this can be found on the TM vision manual.

Now it is needed to add an External Detection block, it can be done by clicking on the icon of the coordinate system with the map marker. Once the block is added it can be open.
Here it is needed to connect the `HTTP`. To do so, open the setting button and insert `http://ip address:port number/api/` in both the `GET` and `POST` slots. It is also needed to define a (key,value) pair for the `POST` method.

Here the `GET` method is just used to test if the server is correctly working. However it is needed to implement this method in the server as it will be shown in 3.2. To correctly finishing set up the node it is needed for the `POST` to be actually able to send back the json file described at the beginning of this chapter. At this point however a json with fake information will work as well.

Once the node has checked that the server is connected and responds correctly, it is possible to move on to the last block which also enhances the quality of the output image.

Finally it is possible to save and exit the node.

3.2 HTTP server

For this task it is needed an external `HTTP` server. Since the robot is connected to the external device through an ethernet cable an does not have access to Internet, a simple `Flask HTTP` server has been chosen to fulfill this purpose.

A Flask server is defined as a server software which is capable of running `HTTP` requests. Flask is a Python library which has a built-in server capable of handling such requests coming from one or multiple devices. However this type of server is single-threaded and hence can handle only one request at a time. Which for the current framework does not constitute a problem.

A Flask server is also capable of:

- **Handling static files** such as Javascript or CSS;
- Storing data in **Flask sessions**. It holds the data temporarily in a temporary folder mapped to a specific ID;
- **Uploading files**;
- **Sending Form data**: The form in HTML collects the information of the required entries and then are forwarded and stored on the server.

However before starting the server, it is needed to perform the stereo calibration because within the `POST` request the object recognition algorithm is also executed. The instruments to undistort and rectify the received images are obtained through the *Stereo Calibration* process.

3.2.1 Stereo calibration

In order to be able to perform depth estimation, it is necessary to perform a stereo calibration of the cameras. In particular this process returns two important instruments:

- **Stereo Maps** that are needed to undistort and rectify images;
- The **two camera matrices** needed to build the intrinsic matrix in equation 2.1.

The procedure consists into taking two pictures at the same time of several different poses of a calibration chessboard. These two pictures normally would be taken from two cameras with different perspective and synced. However in this context only one camera is available mounted on the manipulator.

This setup allowed to have the robot arm going back and fourth between the two predetermined camera positions with precision. The pose of the calibration chessboard would only be changed after two photos of the same pose from the two perspectives were taken. A **minimum of twenty photos** is recommended where each pair of photos showcases a different pose of the chessboard.

After the photos are taken, a script to run the stereo calibration is needed. The code for this process is presented in section A.1. Built-in methods of the `OpenCV` library are used.

Firstly the size of the chessboard and the frame are taken. In this case the chessboard has 9 square corners in the horizontal direction and 6 in the vertical direction. The frame size is 2592x1944 pixel.

After having initialized the arrays that will contain 3D object points and 2D image points, a for loop is used to take each pair of photos and manipulate them. In particular, for each pair of pictures the chessboard corners are found and their coordinates are stored using the `cv2.findChessboardCorners()` method.

This function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners are found and they are placed in a certain order (row by row, left to right in every row). Otherwise, if the function fails to find all the corners or reorder them, it returns 0.[5]

Afterwards the `cv2.cornerSubPix()` finds the sub-pixel accurate location of the corners. This improves the accuracy of the pixel location of the corner in the image. Such subpixel coordinates are stored in the 2D image points object initialised outside the loop.

The whole procedure is repeated for each pair of photos.

The `cv2.calibrateCamera` function finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern. The coordinates of 3D object points and their corresponding 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points, such as a chessboard.[5]

The `cv2.getOptimalNewCameraMatrix` function the camera matrices for each camera pose is obtained. Of course in this context the camera is the same so its intrinsic parameters are exactly the same, however this procedure has been respected even in this setup.

The `cv2.stereoCalibrate` and `cv2.StereoRectify` methods provide the essential matrix, the fundamental matrix and the project matrix which are used to build the stereo maps by using the `cv2.initUndistortRectifyMap` function.

In fact this function computes the joint undistortion and rectification transformation and represents the result in the form of maps for the `remap` function that allows to apply the maps to transform the input image. The undistorted image looks like original, as if it is captured with a camera using the camera matrix equal to the camera matrix output of `cv2.newCameraMatrix` and zero distortion. In case of a monocular camera, the new Camera matrix is usually equal to the original camera matrix, or it can be computed by `cv2.getOptimalNewCameraMatrix` for a better control over scaling.[5]

Finally the camera matrices and the stereo maps are stored in a `.txt` file and in an `.XML` file respectively. The first will be used to build the intrinsic matrix and the latter will be used to undistort and rectify the images received from the camera.

3.2.2 Server structure

The server script is structured as follows:

1. After the imports of the relevant libraries, some constants, the paths for the upload image folder and for the optional `HTML` templates are present. The stereo maps are loaded at this point as well;

2. The `Flask` object is initialized at the top of the script. At the bottom of the script the method `serve` starts the server, taking as input the Flask object, the port number and the name of the host;

3. The error handler attribute for the Flask object allows the server to respond to diverse error messages without stopping the server. It is also possible to set up web pages for the various type of errors (i.e. 404-not found...);

4. A few utility functions are implemented, such as the `UndistandRect` method that uses the previously loaded stereo maps to undistort and rectify images through the built-in OpenCv function `remap`;

5. It is implemented a basic `GET` method that upon a successful GET request, responds with a json containing a positive message. Otherwise it answers with a json file containing a failure message. It is used only for setting up the Vision Node as explained in section 3.1.3;

6. The `POST` method is implemented in order to retrieve the images from the camera sent as a file in the received `POST` request from the `TMFlow`.
   Afterwards the `ORB` algorithm is executed. In particular it is executed on a training image of the object to identify and the image just received from the camera. In a similar way to what has been done in figure 2.5 the algorithm will recognize the target object. Subsequently the square around the object is drawn.
   The attributes of this square (such as its center, its width and it height) is stored inside a json file respecting the format:

```
result = {
    "message" : "success",
    "annotations" : [
            {
                "box_cx": c_x,
                "box_cy": c_y,
                "box_w": width,
                "box_h": height,
                "label": object label,
                "score": 1.0,
                "rotation": rotation°
            }
    ],
    "result": "Image" + filename
}
```

This json file is then converted into a table and stored into a `.txt` file for Matlab to use later.
Finally the json file is sent back to the TMFlow software, ending the `POST` routine and setting the server in idle state.

3.3 Matlab Interface

In this section it is presented the Matlab interface developed in the context of the thesis. It concerns mainly the new matlab class *TM* and some guidelines on how to use this class since it involves establishing a separate client-server relationship between matlab and TMFlow. It is also described the communication protocol used by the Listen Node to communicate with the external devices.

3.3.1 Communication protocol

Before translating each robot command into a Matlab function, it is important to understand how such commands are encapsulated, how they are sent as a data packet to the `TMFlow` program and how received packets should be unpacked.

There are three kinds of possible packet that can be exchanged between the two devices:

- **TMSCT packets** are the basic packets used to exchange information and commands between robot and external device. They can encapsulate all the built-in commands in the script language form;
- **TMSTA packets** are the packets used either by the robot or the external device to acquire statuses or properties. The package format changes based on which type of information is inquired;
- **CPERR packets** are the error packets.

All packages respect the following format:

Start byte	**Head. type**	Sep	**Data Len.**	Sep	**Data**	Sep	*	**Checksum**	**End Bytes**
$	Header	,	Length	,	Data	,	*	Checksum	**\r\n**

Table 3.1: General packet format.

The header type can be one of the three possible kinds (**TMSCT**, **TMSTA** or **CPERR**), the length refers to the length in bytes from the header to the third comma separator included.

The checksum is evaluated by applying an exclusive OR (XOR) of the packet on the same range of the data length.

A short overview of the three kinds of packet is given hereafter.

The **TMSCT packets** are used to exchange commands and acknowledgement packets between robot and external device. The packet respects the format presented in table 3.1, with the Data field subdivided into:

ID	**Sep.**	**Script**
Script ID	,	Script Language

Table 3.2: TMSCT data sub packet format

The Script ID is an alphanumeric byte that is used to identify the particular script. With this ID, the script is matched with its return message.

In the script field it is contained the commands in script language form, following the syntax presented in the *Expression Editor and Listen Node* Manual. It is possible to fit multiple commands in one packet by separating each command with the characters **\r\n**.

The **TMSTA packets** follow the packet format presented in 3.1 but with the Data field subdivided as follows:

Sub cmd.	...	...

Table 3.3: TMSTA data sub packet format

The statuses or properties to acquire depend on the number contained in the `SubCmd` field. In particular:

- **Subcmd 00** is reserved to check if the system is in script control mode or not, which means checking whether the flow has entered the Listen Node or not;

- **Subcmd 01** is reserved to check whether the `QueueTag` numbering is complete or not. The `QueueTag` numbering command is used to concatenate several commands, for example when piling several commands in one script to send in one package. The `PVT` command is an example of command that internally uses these tags. Hence these **TMSTA** packets will hold the tag number to match the command with the same tag. Their `Subcmd` field is subdivided in:

SubCmd		**Tag Number**		**Status**
01	,	01...15	,	true/false/none

Table 3.4: SubCmd field for SubCmd 01.

The Status field is not present if the message is sent from the external device;

- **SumCmd 90...99** is reserved for the robot to send various data and its SubCmd field is subdivided in:

SubCmd		Data
90...99	,	...

Table 3.5: SubCmd field for SubCmd 90...99

The **CPERR packets** are the packets send by the robot to the external device to notify errors. The **CPERR** respects the packet format of table 3.1 and its data field contains:

Error code
Code (00...FF)

Table 3.6: CPERR data sub packet format

The possible code errors are:

- **00** is reserved for no packet error;
- **01** is reserved for general packet error;
- **02** is reserved for checksum field error;
- **03** is reserved for header field error;
- **04** is reserved for data field error;
- **F1** is reserved to the occasion in which the flow has not entered the Listen Node.

3.3.2 TM class

In order to send the commands to the robot in `Script Language` it is necessary to respect the syntax of the robot commands and encapsulate such commands into `TMSCT` packets.
To this end a Matlab class is built where each Matlab function corresponds to a robot command. The full code can be found in the appendix A in the code section A.3. The functions present in the class can be divided in:

- Constructor function;
- Functions to clear the robot buffer and exit the Listen Node;

- Utility functions to pack and unpack the data packets;
- Functions to retrieve and send variables;
- Functions to read and write the digital inputs and outputs;
- Absolute movements functions;
- Relative movements functions.

The constructor function `TM(ip, portControl)` is used to initialize the TCP client object and establish the connection between the external device and the robot. It takes as input the IP address and the port number of the host. It does not return any output. In this case IP = 10.10.10.160 and port number = 5890 have been used.

The `stop(TM)` function is used to stop the robot buffer and flush it. It takes as input the TM object.

The `disconnect(TM)` function is used to exit the Script mode in the robot program. This implies exiting the Listen Node on the robot side. This command has to be used once the communication with the robot is finished and when the robot needs to move on in the TMFlow script. The function takes as input the TM object. Both these functions do not produce any output. If the program features looping and consecutive robot connection through the listen node, it is **very important** to clear the TM object once the current loop communication is ended. The `disconnect(TM)` function does **not** clear the TM object at the moment.

The `writepack(TM,head,command)` function is used to encapsulate the command given as an input in string format. It takes as inputs the TM object and a string containing the header of the packet to be written. At first the function converts the command and the header into an unique string named `unchecked`, respecting the packet protocol presented in 3.3.1.
Afterwards the function evaluates the checksum by using the `checksum(TM, string)` utility function. Then the $ character, the * character and the checksum value are packed together with the `unchecked` string. Finally the terminator characters are added at the end of the packet and returned as the output of this function.

The `checksum(TM, string)` is the utility function that takes as input the TM object and the string on which the operation will be executed. It returns the value resulted from the bit-wise XOR of the message given as input. In order to do this the message is converted from string to double. Then the bit-wise XOR is executed. Finally the result is converted into hexadecimal and returned from the function.

The `read_message(TM, msg)` is an utility function used to unpack only the data field from the packet received from the robot and returns it. It takes as inputs the TM object and the received message.

Firstly the unction finds how many \$ characters are present in the message. This is because often times the packet containing data and the acknowledgement packet are read and stored in the same Matlab variable. This event leads to several misinterpretation of the packet if not taken care of. Hence more than one \$ character means more than one packet are present in the same Matlab variable.

If the message contains only one packet, then it finds the index position of the * character and selects the message from the \$ character up to the last character of the Data field.
Otherwise it finds the index position of either the \$ character and of the * character. The packet that contains the relevant data arrives always first hence the message of interest will be contained from the index position of the first \$ character up to the index position of the first * character minus 2 (because of the comma separator to be excluded from the data to be read).

Finally the remaining message is converted to double and returned from the function. It is here converted to double because this utility function is called from the `ask(TM,id)` function where the message is further processed.

The `ConvertCoord(TM,A,type)` is an utility function that takes the TM object, the data contained in the packet received from the robot (processed from the `read_message(TM, msg)` and in double format) and a type variable. The type variable depends on which kind of information contains A: if $type =' joint'$ then A contains the angles of the robot joints; if $type =' cartesian'$ then A contains the cartesian pose of the robot flange.

Each four bytes of A contains a number and hence A is divided in groups of 4 bytes. Each byte and each division is then converted from double to uint-8 format and then from char to single format. Finally the numbers are piled up in a vector and returned from the function.

The `ask(TM,id)` function takes as input the TM object and an id number. It is used to retrieve four types of specific variables based on the id input:

id = 1 : corresponds to the command used to obtain the six angles of the robot's joints in degrees;

id = 2 : corresponds to the command used to obtain the 3D position of the robot flange with respect to the *Robot frame* and the three angles for the orientation of the flange, in degrees;

id = 3 : corresponds to the command used to obtain the composite force of the flange over x, y and z axes;

id = 4 : corresponds to the command used to obtain the composite speed of the flange over x, y and z axes.

This division is implemented with a switch statement controlled with the id number given as input. Each switch case is structured as follows:

1. String composition of the robot command to be encapsulated;

2. Encapsulation of the package through the `writepack(TM,head,comando)` utility function;

3. Sending the encapsulated packet through the built-in Matlab function `write(connection,data,fromat)`;
4. Waiting for the acknowledgement message;
5. Unpacking of the received message through the `read_message(TM, msg)` utility function;
6. Conversion of the received message into single type either directly or through the `ConvertCoord(TM,A,type)` utility function.

In order to have a reactive function, a callback function has been implemented inside the `ask(TM,id)`. The callback function is triggered every time a **\r\n** character is read. This allows to receive more efficiently the incoming packets. Before adding this feature often it would be caught only the acknowledgements packet and not the packet containing relevant data. With this strategy both the packets are received.

Once the packets are received, they are filtered through the `read_message(TM, msg)` function and eventually from the `ConvertCoord(TM,A,type)` function.

The `ask(TM,id)` function finally returns the inquired information in single format.

The `ReadIO(TM, source, type, id)` and `WriteIO(TM, sourec, type, id, value)` functions are used to respectively read and write on the inputs or outputs. It is not possible however to write on the inputs, it's only possible to read them. The read function returns the read value while the write function does not return any output.

The two functions take as inputs:

- The TM object;
- The source of the I/O: it can be either from the Control box or the End Module. Hence the two possible sources are either $'ControlBox'$ or $'EndModule'$;
- The type of I/O: it can be input or output, analog or digital. The four possible types are AI, AO, DI and DO;
- The id number of the selected input or output;
- The write function takes as input the value to be written on the selected output.

These two functions have also an `Instant` version of themselves, becoming `InstantReadIO(TM, source, type, id)` and `InstantWriteIO(TM, source, type, id, value)`. The difference is that in the case of the first pair the commands are executed respecting the sequence of commands that the robot has received. Instead with the instantaneous commands, they are executed as soon as the robot receives them.

The movement commands are divided into absolute and relative movement commands. However they all have almost the same type of inputs. The variables taken as inputs from the functions are:

- TM object;
- **`txt`**: it is a character string that represents the type of coordinates that are given, the format of the speed and the format of the movement blending. Each movement as a different set of possible **`txt`** variable. It is always composed by three uppercase character which have different meaning for absolute and relative movements. For absolute movements:

Motion format : `C` for Cartesian pose or `J` for joint pose;

Speed format : `P` for speed expressed as percentage or `A` for speed expressed in $[mm/s]$;

Blending format : `P` for blending expressed in percentage or `R` for blending expressed in radius.

For relative movements:

Motion format : `C` for pose expressed with respect to current base, `T` for pose expressed with respect to tool base or `J` for pose expressed in joint angles;

Speed format : `P` for speed expressed as percentage or `A` for speed expressed in $[mm/s]$;

Blending format : `P` for blending expressed in percentage or `R` for blending expressed in radius.

- **coord**: the input coordinates either in Cartesian or joint angle format;
- **vel**: speed value. Either in mm/s or in percentage;
- **Ta**: maximum time interval to reach maximum velocity;
- **racc**: Percentage for blending movements;
- **PrecPos**: boolean variable. If it is set to true, the precision positioning is disabled and it is enabled if the variable is set to false.

The set of absolute movements are:

- `PTP(obj,txt,coord,vel,Ta,racc,PrecPos,conf)`: this command executes a point to point movement starting from the current pose of the robot and reaching the pose given as input. This type of movement determines the robot's motion by calculating the angular variation of each axis. It takes the shortest motion possible to reach the end point. The `conf` variable is used to set the robot pose, following the diagram shown in fig.3.1. This input variable is optional.
  The input vector `conf` takes the form of 0-1,2-3,4-5 where:

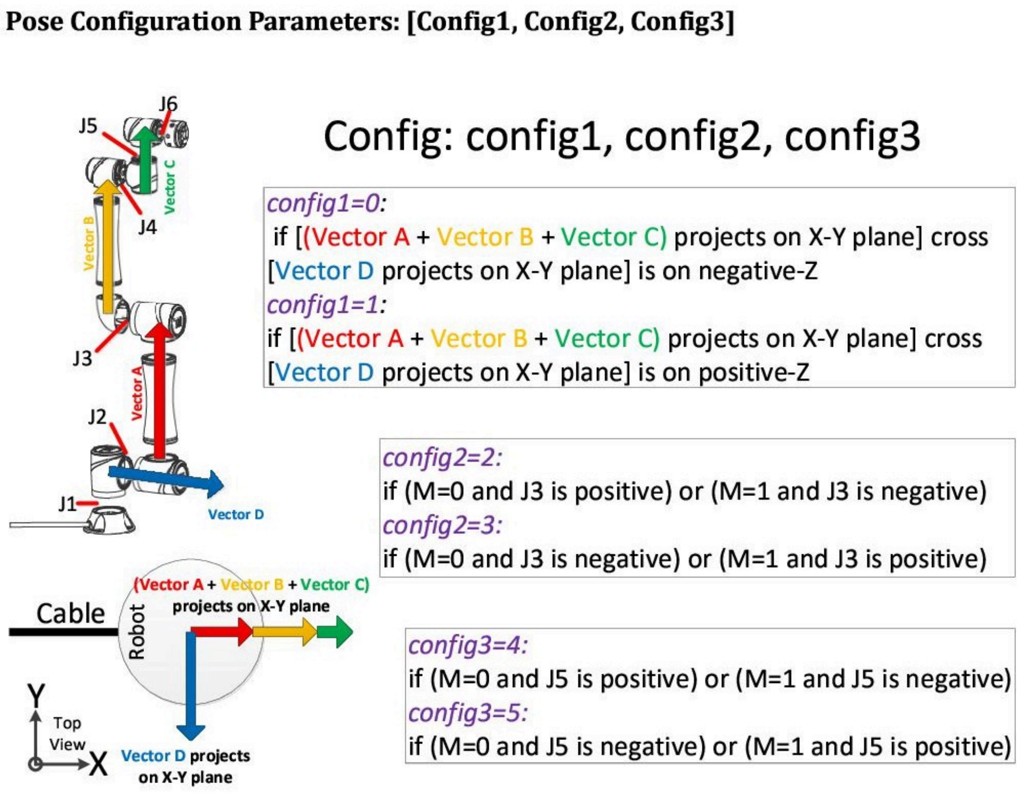
  0 = Right arm configuration;

  1 = Left arm configuration;

  2 = Upward elbow configuration;

  3 = Downward elbow configuration;

  4 = Non flipped wrist configuration;


Pose Configuration Parameters: [Config1, Config2, Config3]

Config: config1, config2, config3

config1=0:
if [(Vector A + Vector B + Vector C) projects on X-Y plane] cross [Vector D projects on X-Y plane] is on negative-Z
config1=1:
if [(Vector A + Vector B + Vector C) projects on X-Y plane] cross [Vector D projects on X-Y plane] is on positive-Z

config2=2:
if (M=0 and J3 is positive) or (M=1 and J3 is negative)
config2=3:
if (M=0 and J3 is negative) or (M=1 and J3 is positive)

config3=4:
if (M=0 and J5 is positive) or (M=1 and J5 is negative)
config3=5:
if (M=0 and J5 is negative) or (M=1 and J5 is positive)

Figure 3.1: Possible robot poses.

5 = Flipped wrist configuration;

The possible options for `txt` variable are: `JPP` or `CPP`.

- `Circle(TM,txt,puntomezzo,puntofin,speed,Ta,racc,ArcAngle,PrecPos)`: this command executes a circular movement starting from the current pose of the robot, passing through the input variable `puntomezzo` and reaching the destination `puntofin`. If non-zero value is given to the `ArcAngle` variable, the TCP will keep the same pose and move from current point to the assigned arc angle via the given pose and end pose on arc. If zero is given, the TCP will move from current pose to end pose via the point on arc with linear interpolation on pose.
The possible options for `txt` variable are: `CAP` or `CPP`;

- `Line(TM,txt,coord,speed,Ta,racc,PrecPos)`: this command plans a straight path for the robot flange. It starts from the current robot pose and end on the input end pose. The possible options for `txt` variable are: `CPP`, `CAP`, `CPR` or `CAR`;

- `PLine(obj,txt,coord,vel,Ta,racc)`: it is a command similar to `Line` but different settings for movement blending. It is particularly useful when treating paths with waypoints;

The possible options for `txt` variable are: `JAP` or `CAP`;

- `PVT(TM ,ps ,vs ,coord , duration)`: this matlab function is used to exploit the robot PVT mode. This modality allows to plan complex robot paths with multiple points, each with possibly different speed, acceleration and blending settings. This function encapsulates three robot commands: `PVTEnter()`, `PVTPoint()` and `PVTExit()`. `PVTEnter()` and `PVTExit()` are used to start and to end the PVT robot mode. The `PVTPoint()` is used to specify poses, speed, acceleration (in terms of time to reach maximum velocity) and blending.
  In the corresponding Matlab function is possible to give as inputs:

coord_type : it can be set to either `J` or `C` which respectively stand for Joint and Cartesian;

ps : a matrix that contains a different pose for each row in the form of x,y,z,Rx,Ry,Rz;

vs : a speed vector that contains a specified speed for each row (and hence for each point). Speed is specified in mm/s;

duration : a vector containing in each row a specified `Ta`. Each time variable is specified in milliseconds.

The set of relative movements are:

- `Move_Line(TM,txt,coord,vel,Ta,racc,PrecPos)`: `txt` takes as possible values `CPP`, `CPR`, `CAP`, `CAR`, `TPP`, `TPR`, `TAP` or `TAR`;
- `Move_PLine(TM,txt,coord,vel,Ta,racc)`: `txt` takes as possible values `CAP`, `TAP` or `JAP`;
- `Move_PTP(TM,txt,coord,vel,Ta,racc,PrecPos)`: `txt` takes as possible values `CPP`, `TPP` or `JPP` ;

These commands implement the corresponding type of movement in the same way that they were implemented for the absolute movement, despite the fact that they are relative movements.

Chapter 4

Final Experiment

4.1 Techman TM5-700

This thesis is developed in the context of employing cooperative mechanical arms for high precision processes. Robots are typically confined in a dedicated cell that is closed to human operators.

Cooperative robots are designed to work in parallel with a human operator and hence they frequently do not require an enclosing cell for their workspace.

4.1.1 Manipulator specifications

The `TM5-700` is a 6 axis compact `cobot`(cooperative-robot) designed to fit into production lines. It is provided with a built-in vision system thought to satisfy the needs of small parts assembly, production processes in electronics and consumer goods.

It belongs to the cooperative robots family and hence it is provided with sensors to prevent impact with the human operator. It is possible to tune the safety stops by tightening or loosening the torque thresholds of each joint.

The robot is mounted on a moving table and is not contained in a cell. It is equipped with an integrated operative system called `TMFlow` that is discussed in 3.1.
The datasheet for this robot is provided in Appendix B in fig.B.2.

4.1.2 Gripper Notes

For this application a two finger pneumatic gripper is used. It is connected to the robot's control box via an electromagnetic valve. The gripper specifications are presented in Appendix B in fig. B.3 under the column VR16-60.

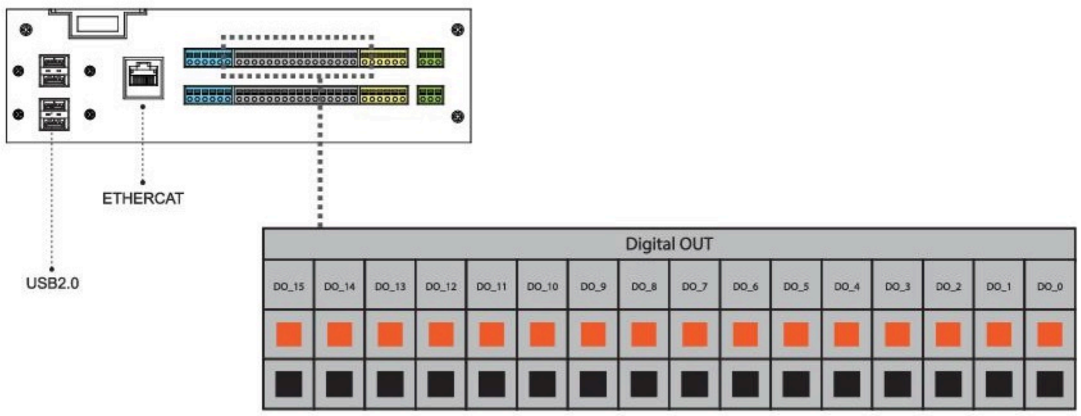
The gripper is controlled through a digital output that can also be controlled by the matlab interface 3.3.

Value	Effect on gripper
0	open
1	closed

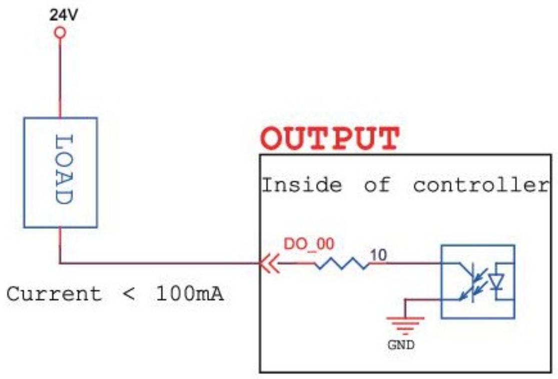
Table 4.1: Notation used to handle the gripper.

The notation showed in the table 4.1 can be reversed if the electrovalve is left open when the system is started. So extra attention to the gripper is needed when turning on and off the system.

In Fig.4.1 is shown the electrical scheme used to attach the gripper to the control box.



(a) Electrical interface of the control box.



(b) Electrical scheme.

Figure 4.1: Electrical connection used.

4.2 Integrating the framework's components

4.2.1 Flowchart of the whole process

Before starting the framework it is given that the Stereo Calibration is already executed. Hence the stereo maps already exist and will be loaded in the `HTTP` server script just before starting the flask application. The whole process is schematically described referring to the flowchart in figure 4.2. The full code of each system is included in Appendix A.

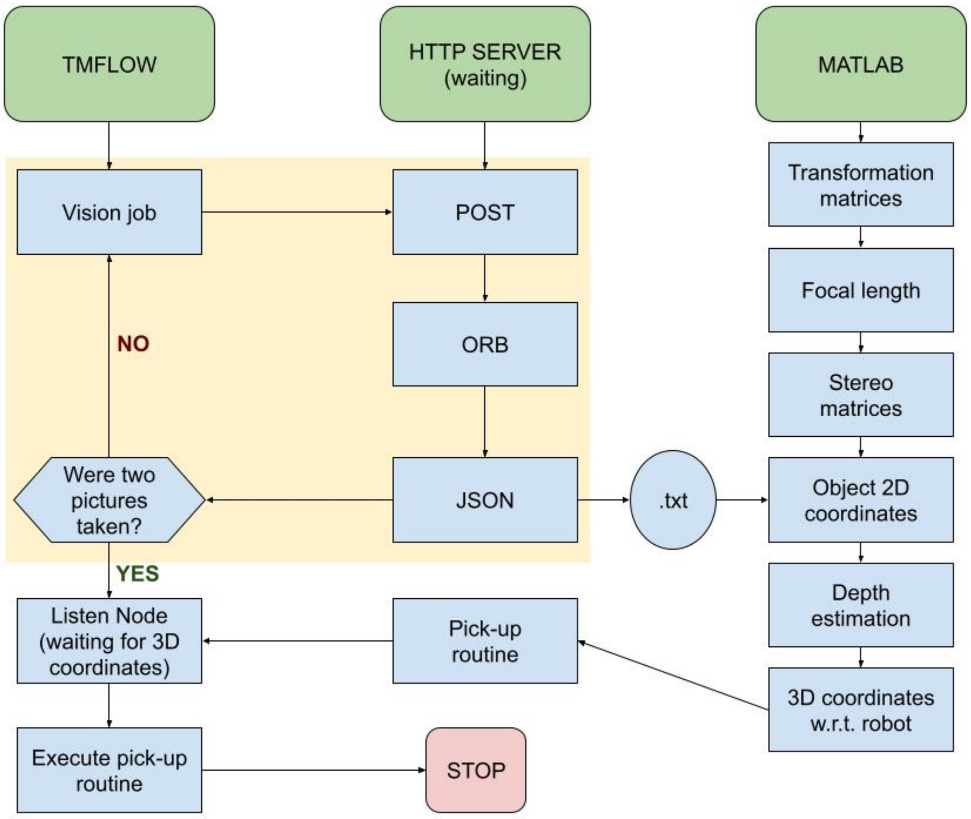


Figure 4.2: Flowchart of the framework logic

Once the `HTTP` server is started it can be left running in the background on the external device. It will wait in idle state for the incoming requests.

The TMFlow program can now be started. The logic of the program is shown in 4.3.

The robot software after positioning the manipulator on the first camera position, it will enter the first **Vision Job** node. The camera takes the picture and sends a `POST` request to the `HTTP` server containing the image. This request will also carry the (key,value) pair as: `(model_id, sx)`. The picture is taken at height equal to 400 mm for the robot flange. In other words the flange is

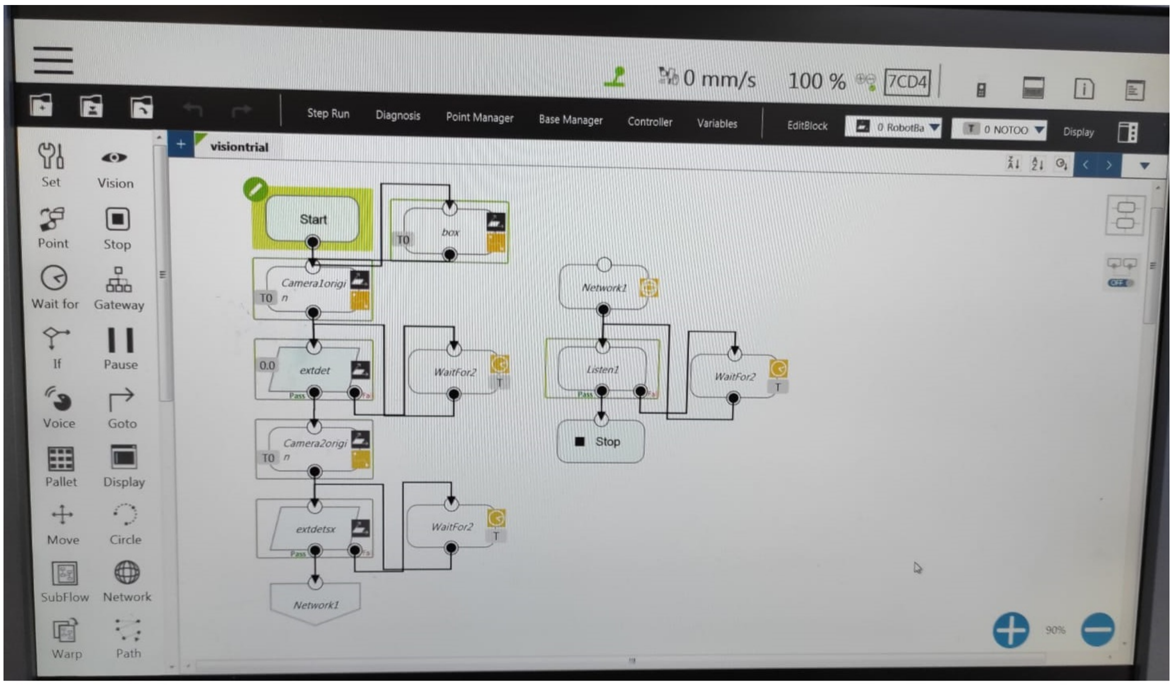


Figure 4.3: TMFlow program.

at height 400 mm from the table surface. The camera results at height 347,8 mm from the table surface. The measured distance between camera and object is 286 mm. This flange height was decided with regards to the fact that the maximum working distance of the camera is 300 mm as reported in section 2.3.1.

Inside the `POST` function of the server script, the received picture will be saved onto the external device with the name composed by the upload folder path plus `image_SX` upon recognizing the pair (key,value) as (model_id, sx).

Subsequently the `ORB` algorithm is executed. It takes as image inputs a training image previously loaded that contains only the object to be identified and the photo received from the camera. The `ORB` subroutine is shown here:

```
#— ORB
#IMPORTANT_NOTE: avoid irrelevant corners in query pictures at all costs!!

outputimg = queryim.copy()
width, height = Image.open(UPLOAD_FOLDER + r"\image_DX.jpg").size
#widthxheight pixels of query img

# OpenCV uses BGR as its default colour order for images, matplotlib uses RGB.
# So if plt is used then uncomment this conversion
# trainim = cv2.cvtColor(trainim, cv2.COLOR_BGR2RGB)
# queryim = cv2.cvtColor(queryim, cv2.COLOR_BGR2RGB)

# Initiate ORB detector
orb = cv2.ORB_create()

# find the keypoints and descriptors with ORB
kp1 = orb.detect(trainim,None)
kp2 = orb.detect(queryim,None)

kp1, des1 = orb.compute(trainim, kp1)
kp2, des2 = orb.compute(queryim, kp2)

# create BFMatcher object
bf = cv2.BFMatcher.create(cv2.NORM_HAMMING, crossCheck=False)

# Match descriptors, knn method.
matches = bf.knnMatch(des1,des2,k=2)

# I can also mask the keypoints by "filtering" only the best
# ones; a.k.a. the keypoints whose descriptor
# have low hamming distance

# Sort them in the order of their distance.
matches = sorted(matches, key = lambda x: x[:][1].distance)

good = []
for m,n in matches:
    if m.distance < nu*n.distance:
        good.append([m])

Matched = cv2.drawMatchesKnn(trainim,kp1,queryim,kp2,
    good,outImg=None,matchColor=(0, 155, 0),
    singlePointColor=(0, 255, 255),matchesMask=None,flags=0)

kp3 = [] #creating an empty keypoint object

for i in range(len(good)):
    a = good[i][0].trainIdx
    idx=kp2[a].pt
    key = cv2.KeyPoint(idx[0],idx[1],1)
    kp3.append(key)


output_img = cv2.drawKeypoints(outputimg, kp3 ,outputimg,
    (255,0,0),flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)
```

The `ORB` returns matched keypoints and descriptors of the two photos producing a `match` object. It has been selected the Hamming distance as distance criteria and the k-nearest neighbour(`knn`) modality. The `knn` modality implies that each keypoint is matched with the best k matches. In this case k = 2.

As explained is section 2.2, the descriptors are binary strings containing orientation information of pixel patches. And so the matches are sorted with respect to their hamming distance. Afterwards they are further filtered in order to select only the closest pairs of keypoint and descriptor and hence those pairs with low Hamming distance.

A short subroutine is used to **localize** the object. In particular from the `match` object two separate pixel coordinates arrays are produced through a for loop as shown in the following script:

```
#-- Localize object
obj = np.empty((len(good),2), dtype = np.float32)
scene = np.empty((len(good),2), dtype = np.float32)
output = np.empty((len(good),2), dtype=np.float32)

for i in range(len(good)):
    obj[i,0] = kp1[good[i][0].queryIdx].pt[0]
    obj[i,1] = kp1[good[i][0].queryIdx].pt[1]
    scene[i,0] = kp2[good[i][0].trainIdx].pt[0]
    scene[i,1] = kp2[good[i][0].trainIdx].pt[1]
# kp3 is used to create an output image with training image
# and scene image one next to the other
for i in range(len(kp3)):
    output[i,0] = kp3[i].pt[0]
    output[i,1] = kp3[i].pt[1]
```

One array is for the training image and one for the scene image. In the match object of the training image are stored the i-th (x,y) coordinates (in pixel) of the training image which correspond to the i-th (x,y) coordinates of the scene image. For the scene coordinate array is done the opposite.

With these two coordinate sets is possible to build an Homography matrix. An Homography matrix is a 3-by-3 matrix that represents the perspective transformation from the training image to the scene object.

After saving the corners of the training image (and hence of the object to detected), the Homography matrix is used to map the training image corners onto the scene image. A box around the detected object is drawn by using a method that takes as input two points and draws a line. The two points are pairs of the mapped corners onto the scene image, taken two by two as shown here:

```
try:
    H, _ = cv2.findHomography(obj,scene,cv2.RANSAC)
    H2, _ = cv2.findHomography(obj, output, cv2.RANSAC)
except:
    print("Not enough matches or zero matches found!")
    result = {
        "message": "Not enough matches or zero matches found!",
        "result": None
    }
    return result #with this return the server goes back to idle state

#-- Get the corners from the training image
obj_corners = np.empty((4,1,2), dtype=np.float32)
obj_corners[0,0,0] = 0
obj_corners[0,0,1] = 0

obj_corners[1,0,0] = trainim.shape[1]
obj_corners[1,0,1] = 0

obj_corners[2,0,0] = trainim.shape[1]
obj_corners[2,0,1] = trainim.shape[0]

obj_corners[3,0,0] = 0
obj_corners[3,0,1] = trainim.shape[0]

#-- Here I get the corners of the train object "mapped" on to the query image
# coordinates through the homography matrix
# previously evaluated
scene_corners = cv2.perspectiveTransform(obj_corners, H)
output_corners = cv2.perspectiveTransform(obj_corners, H2)

#-- Draw lines between the corners (the mapped object in the scene - image_2 )
cv2.line(Matched,
(int(scene_corners[0,0,0] + trainim.shape[1]), int(scene_corners[0,0,1])),\
(int(scene_corners[1,0,0] + trainim.shape[1]), int(scene_corners[1,0,1])),
(0,255,0), 4)
cv2.line(Matched,
(int(scene_corners[1,0,0] + trainim.shape[1]), int(scene_corners[1,0,1])),\
(int(scene_corners[2,0,0] + trainim.shape[1]), int(scene_corners[2,0,1])),
(0,255,0), 4)
cv2.line(Matched,
(int(scene_corners[2,0,0] + trainim.shape[1]), int(scene_corners[2,0,1])),\
(int(scene_corners[3,0,0] + trainim.shape[1]), int(scene_corners[3,0,1])),
(0,255,0), 4)
cv2.line(Matched,
(int(scene_corners[3,0,0] + trainim.shape[1]), int(scene_corners[3,0,1])),\
(int(scene_corners[0,0,0] + trainim.shape[1]), int(scene_corners[0,0,1])),
(0,255,0), 4)

cv2.line(output_img,
(int(output_corners[0,0,0] ), int(output_corners[0,0,1])),\
(int(output_corners[1,0,0] ), int(output_corners[1,0,1])),
(0,255,0), 4)
# (x coordinate 0,0,0 , y coordinate 0,0,1); other two coords after
# backslash are the x,y of the second point
cv2.line(output_img,
(int(output_corners[1,0,0] ), int(output_corners[1,0,1])),\
(int(output_corners[2,0,0] ), int(output_corners[2,0,1])),
(0,255,0), 4)
cv2.line(output_img,
(int(output_corners[2,0,0] ), int(output_corners[2,0,1])),\
(int(output_corners[3,0,0] ), int(output_corners[3,0,1])),
(0,255,0), 4)
cv2.line(output_img, (int(output_corners[3,0,0] ), int(output_corners[3,0,1])),\
(int(output_corners[0,0,0]), int(output_corners[0,0,1])),
(0,255,0), 4)

#-- Saving outputimage with square
if model_id == "dx":
    output_img=cv2.cvtColor(output_img, cv2.COLOR_BGR2RGB)
    plt.imsave(os.path.join(app.config['UPLOAD_FOLDER'],
        "outputDX_img.jpg"), output_img)
else:
    output_img=cv2.cvtColor(output_img, cv2.COLOR_BGR2RGB)
    plt.imsave(os.path.join(app.config['UPLOAD_FOLDER'],
        "outputSX_img.jpg"), output_img)

#-- Get coordinates, height and width of square box
```

```
cx = (output_corners[0,0,0]+output_corners[1,0,0]+
    output_corners[2,0,0]+output_corners[3,0,0])/4
cy = (output_corners[0,0,1]+output_corners[1,0,1]+
    output_corners[2,0,1]+output_corners[3,0,1])/4
box_h = np.sqrt(np.square(output_corners[0,0,0]-output_corners[3,0,0])+
    np.square(output_corners[0,0,1]-output_corners[3,0,1]))
box_w = np.sqrt(np.square(output_corners[0,0,0]-output_corners[1,0,0])+
    np.square(output_corners[0,0,1]-output_corners[1,0,1]))

#-- Get rotation of square box
theta = - np.arctan2(H2[0,1], H2[0,0])
theta = np.rad2deg(theta)
```

The corners of the drawn box are then used to retrieve the required box parameters such as its center, its width and its height. These information will be packed into the json file to be sent back to the robot.

Afterwards the rotation of the object is evaluated as:

$$\theta = \frac{180}{\pi}(-atan2(\frac{H(0,1)}{H(0,0)})) \tag{4.1}$$

where H is the previously calculated Homography matrix.

Finally the json file is packed, sent back to the **Vision Job** node and saved on a .txt file to be picked up from the Matlab script later.

This procedure is then repeated for the second photo. The distance between the two camera positions is equal to $Baseline = 81mm$.

In figure 4.4 the two camera positions are shown.

After the two vision jobs are executed, the Matlab script can be executed. Inside this script the intrinsic and extrinsic matrices are built as exposed in section 2.1. The focal length is estimated following the procedure described in section 2.3.1.

The (x,y) coordinates of the center of the boxes are extracted from the json variables previously saved in two separate .txt files. They are converted from *Pixel frame* to *Image frame* of each respective view by using the inverse of the Intrinsic matrix K_{IP}.

Now the (x,y) coordinates of the two *Image frames* are projected onto the *Image frame* of the object to be detected by using the equations 2.4 and 2.5. Using these equations actually produce coordinates with respect to the *World frame*. The depth is estimated through the procedure in section 2.3 and it is added as the third coordinate of the point found in this way and is called [x,y,z].

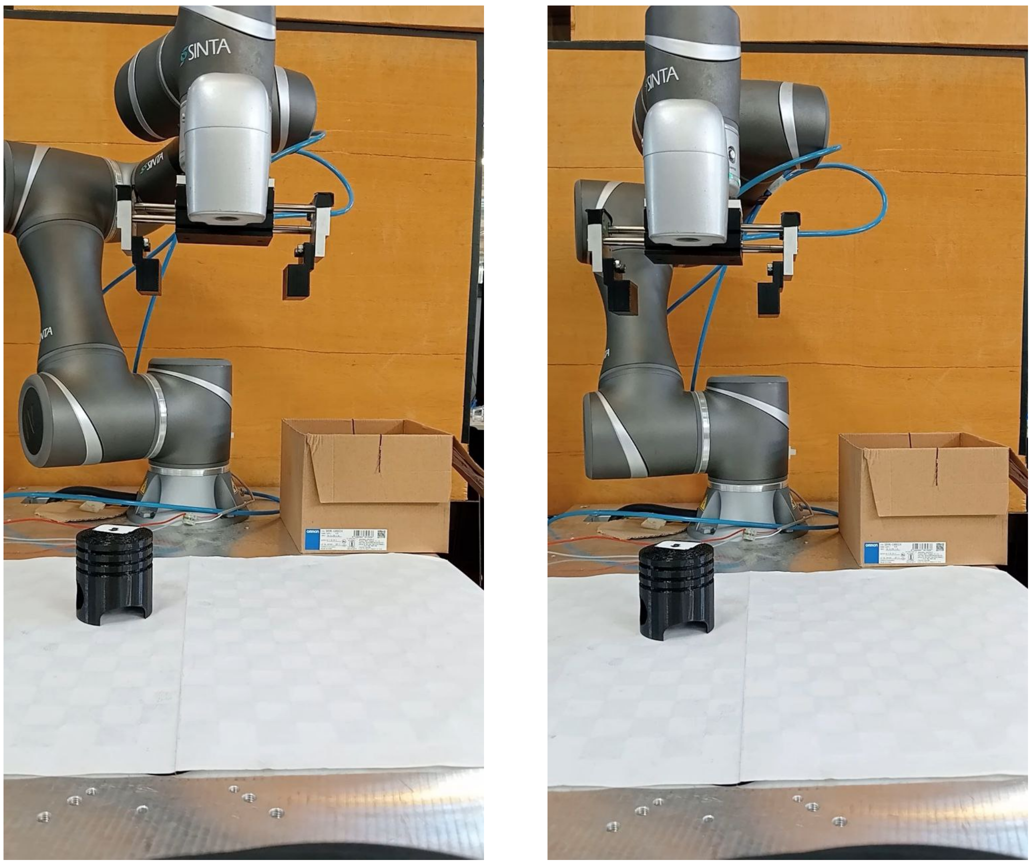

(a) Left view **(b)** Right view

Figure 4.4: Robot camera taking two pictures in the two poses.

This point is now converted from *World frame* to *Robot frame* by using the extrinsic matrix $K_{extrinsic}$ derived in 2.1.2. In practice:

$$Object_{Robot} = T_{CaF} T_{WR} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{4.2}$$

In the meantime the program on TMFlow has reached the Listen Node and it is now ready to receive commands from the Matlab interface.

To evaluate at which height the robot has to position the grip, a few physical measures must be considered. Since the gripper is attached to the flange, the grip and object height parameters must be considered:

Height	Value [mm]
a = Gripper base	60
b = Gripper finger	40
c = Object	8

Table 4.2: Gripper and object parameter list.

In order to not make the gripper crash on the object, it has been considered half the height of the gripper finger.

Hence the resulting height coordinate to position the gripper is given by:

$$height = a + \frac{b}{2} - c \tag{4.3}$$

using the same variables used in table 4.2.

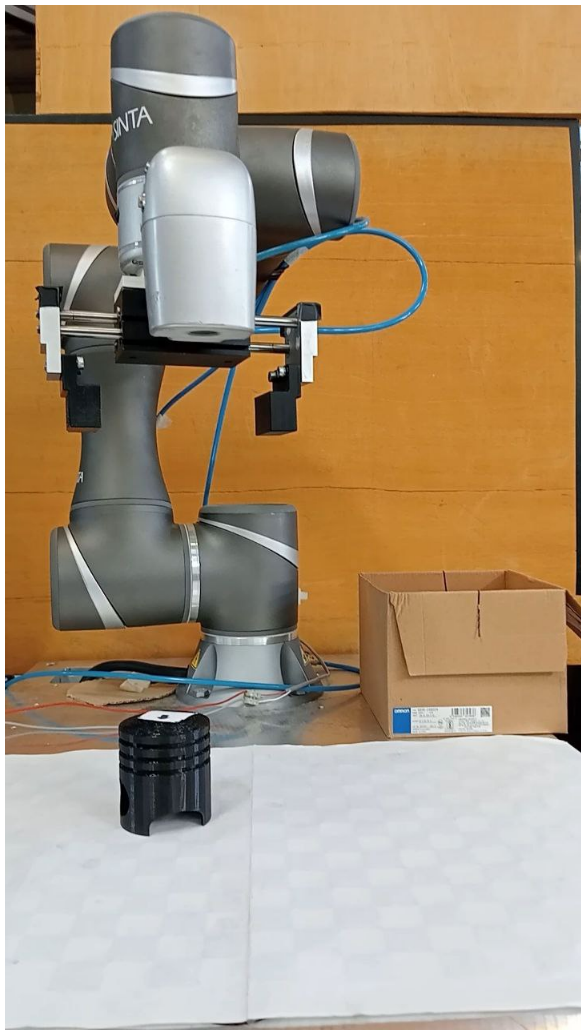
Finally the Matlab script connects to the robot through the TM class and sends the pick up routine. Hereafter is shown the code that implements the robot movement:

```
    %% Robot pick up routine
% %— Connect Robot————————————————————————————————
src = TM('10.10.10.160',5890);
% % —— Move above the evaluated 3D pose ——————————————————
PLine(src, 'CAP',[des2(1),des2(2),des2(3),179,0,0],200,100,100);
%—— Rotate gripper above the object
Move_PLine(src, 'CAP',[0,0,0,0,0,rot],200,100,100);

% %— Open gripper ———————————————————————————
% % high == chiuso 1
% % low == aperto 0

Read = InstantReadIO(src, 'ControlBox', 'DO',0);
pause(0.05)
if  Read == 1
    WriteIO(src, 'ControlBox', 'DO',0,0);
end
pause(0.05)

% %— Descend
PLine(src, 'CAP',[des2(1),des2(2),altezza,179,0,rot],200,100,100);

% %— Close gripper
pause(1)
WriteIO(src, 'ControlBox', 'DO',0,1);
```

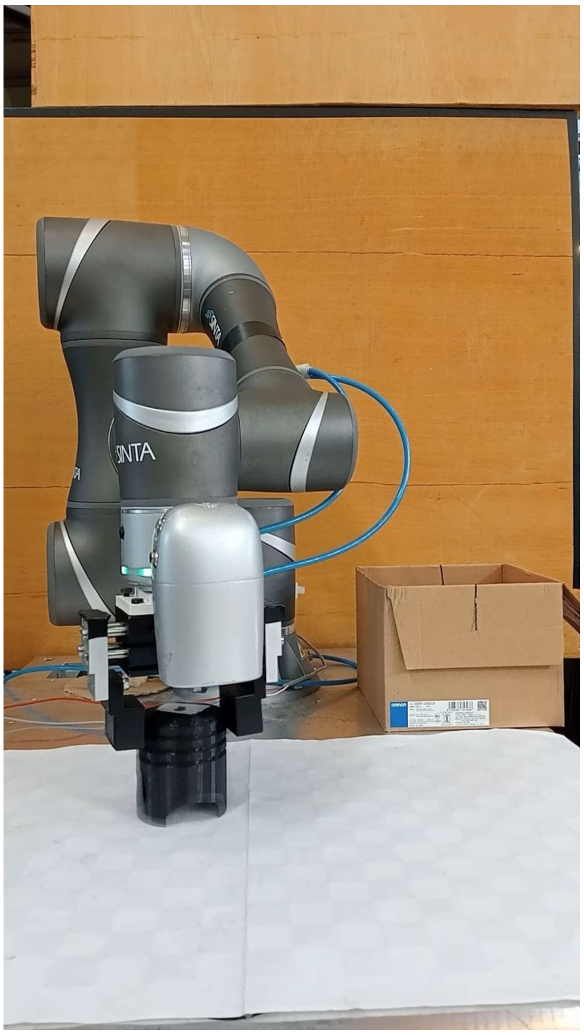
```
% %— Move away
Move_PLine(src, 'CAP', [0,0,des2(3),0,0,0], 200, 100, 100);
pause(5)

% %— Go to cardboard box
PLine(src, 'CAP', [243.44, −105.93, 400, −179,0,90], 200,100,100);
pause(5)

% %— Descend to box
PLine(src, 'CAP', [243.44, −105.93, 230, −179,0,90], 200,100,100);
pause(5)

% %— Let go of object
WriteIO(src, 'ControlBox', 'DO', 0, 0);
pause(2)

% %— Move upwards
PLine(src, 'CAP', [243.44, −105.93, 400, −179,0,90], 200,100,100);
% pause(5)

% %— Disconnect Robot
pause(10)
disconnect(src);
```

Firstly the TCPclient built through the TM class establishes the connection between external device and robot.

The robot moves the gripper above the detected object (and hence the flange is still at height equal to 400 mm). Subsequently the gripper rotates above the object of the quantity evaluated from the POST function at section 4.1 and saved into the jsonfile (fig. 4.5a).



(a) Robot rotates gripper according to previously evaluated object orientation.



(b) Robot successfully grasps the object.

Figure 4.5: Robot orients its gripper and picks the detected object.

Now by reading the Digital output status it is asserted if the gripper is already open or not. If it is closed then it is opened by writing 0 onto the digital output. Otherwise it is left open.

The robot now descends onto the object with the rotated gripper at height equal to 4.3. Once this movement is completed the gripper is closed, properly grabbing the object (fig. 4.5b).

The robot then moves the object upwards and moves to a carboard box placed near the robot and which coordinates are known (fig. 4.6a).

The robot descends with the gripper inside the box, opens the gripper and frees the object from its grasp (fig. 4.6b). The arm now moves away in a straight upward direction (fig. 4.6c).

(a) Robot positions itself on the box while holding the object.

(b) Robot settles into the box and opens the gripper to let go of the object.

(c) Robot moves away from placed object.

The `disconnect(src)` command sends the robot command `ScriptExit()` which exits the Listen Node on the pass Path. Here the TMFlow program stops.

4.2.2 Depth estimation

In order to evaluate the precision of the depth estimation procedure, a series of 30 trials has been conducted.

The target object has not been moved. Each trial is independent from the other because each trial started from fresh values and fresh photos. In Fig.4.7 it is summarized the result:

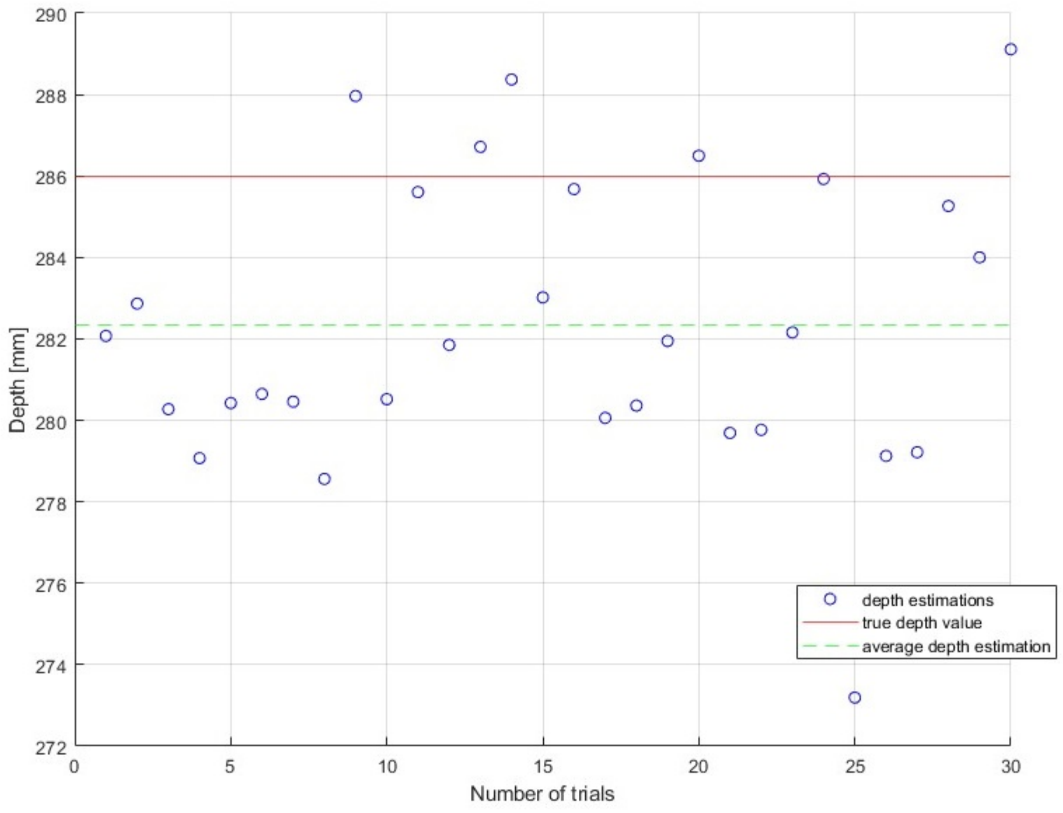


Figure 4.7: Result of consecutive depth estimations.

The results of these trials can be summarized in:

Mean	Standard deviation
282.35 mm	3.55 mm

Table 4.3: Depth estimation mean and deviation standard.

With these values it is demonstrated that the vision system can pick up the detected objects with a good precision.

Chapter 5

Conclusion

This work's objective is to build a framework in which is possible to communicate with a cooperative robot through a Matlab interface. The `TM5-700` robot has an Eye-in-Hand camera mounted on its arm and in this case it is needed an `HTTP` server to retrieve the pictures. Hence it is shown how to build an `HTTP` server with the Flask Python library.

The server is used not only to retrieve the pictures taken from the Eye-in-Hand camera but also to detect a target object present in the pictures' view. To retrieve the correct 3D position it is employed the Binocular vision system model that returns a good estimation of the depth information. Information which is impossible to retrieve from one single photo. Hence two photos are used to apply this model.

The `ORB` algorithm is used for the object detection part. This algorithm returns the (x,y) pixel coordinates of the detected object. This information together with the depth information are translated into coordinates with respect to the *Robot frame*.

Finally a Matlab script is produced utilizing the TM class which represents the interface that communicates with the robot arm. In this script the pick up routine is written and finally the robot successfully grabs the target object.

It is shown that the depth estimation is also quite precise, producing an error of less than 4 mm on thirty consecutive trials.

This ensures the robot capability of successfully picking up the object detected through the camera pictures and manipulated from the `ORB` algorithm and the Binocular vision system model.

The main strength of this work is its adaptability. In fact by adapting the TM class to another robot communication protocol, it is possible to use the same framework for other robotic systems.

Moreover the `ORB` algorithm can be easily replaced by other computer vision algorithms without re-implementing from scratch the `HTTP` server.

Appendix A

Full Codes

A.1 Code: Stereo Calibration

```
import numpy as np
import cv2 as cv
import glob
import pandas as pd

from tabulate import tabulate

chessboardSize=(9,6)


frameSize = (2592,1944)
#— Termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)
#— Prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((9*6,3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)

#— Size of each chessboard square in world coordinates and in mm
objp = objp*21

#— Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpointsL = [] # 2d points in image plane.
imgpointsR = [] # 2d points in image plane.

#— Routes to left and right images
imagesLeft = glob.glob('C:\\Users\\bianc\\TMvision_TmHttp_server_sample_code\\python_example\\images\\calib_trial_400\\Left\\*.jpg')
imagesRight = glob.glob('C:\\Users\\bianc\\TMvision_TmHttp_server_sample_code\\python_example\\images\\calib_trial_400\\Right\\*.jpg')

#— For loop
for imL,imR in zip(imagesLeft,imagesRight):
    #— Pick pair of left and right images
    imgL = cv.imread(imL)
    imgR = cv.imread(imR)

    #— Convert to gray both images
    grayL = cv.cvtColor(imgL, cv.COLOR_BGR2GRAY)
    grayR = cv.cvtColor(imgR, cv.COLOR_BGR2GRAY)

    #— Find the chess board corners
    retL, cornersL = cv.findChessboardCorners(grayL, chessboardSize, None)
    retR, cornersR = cv.findChessboardCorners(grayR, chessboardSize, None)

    #— If chessboard is found, add object points, image points (after refining them)
    if retL and retR == True:
        objpoints.append(objp)

        #— This function iteratively refines the corner locations untill the termination criteria is reached
        cornersL = cv.cornerSubPix(grayL, cornersL, (11,11), (-1,-1), criteria)
        imgpointsL.append(cornersL)
        cornersR = cv.cornerSubPix(grayR, cornersR, (11,11), (-1,-1), criteria)
        imgpointsR.append(cornersR)

        #— Draw and display the corners
        cv.drawChessboardCorners(imgL, chessboardSize, cornersL, retL)
        cv.namedWindow("outputL", cv.WINDOW_NORMAL)
        cv.imshow("outputL", imgL)
        cv.resizeWindow("outputL", 960, 540)

        cv.drawChessboardCorners(imgR, chessboardSize, cornersR, retR)
        cv.namedWindow("outputR", cv.WINDOW_NORMAL)
        cv.imshow("outputR", imgR)
        cv.resizeWindow("outputR", 960, 540)

        cv.waitKey(1000)
    else:
        print("No chessboard found")

cv.destroyAllWindows()

#— Calibration for each camera
retL, cameraMatrixL, distL, rvecsL, tvecsL = cv.calibrateCamera(objpoints, imgpointsL, grayL.shape[::-1], None, None)
newCameraMatrixL, roi_L = cv.getOptimalNewCameraMatrix(cameraMatrixL, distL, frameSize, 1, frameSize)

retR, cameraMatrixR, distR, rvecsR, tvecsR = cv.calibrateCamera(objpoints, imgpointsR, grayR.shape[::-1], None, None)
newCameraMatrixR, roi_R = cv.getOptimalNewCameraMatrix(cameraMatrixR, distR, frameSize, 1, frameSize)

#— Stereo Vision Calibration
flags = 0
flags = cv.CALIB_FIX_INTRINSIC #with this flag intrinsic matrices are fixed so that only
# R,T, fundamental and essential matrices are evaluated

criteria_stereo = (cv.TERM_CRITERIA_EPS+cv.TERM_CRITERIA_MAX_ITER, 30 , 0.001)

retStereo, newCameraMatrixL,distL,newCameraMatrixR, distR, rot, trans, essentialMat, fundamentalMat = cv.stereoCalibrate(objpoints,
        imgpointsL,imgpointsR, newCameraMatrixL, distL, newCameraMatrixR, distR, frameSize)

#— Stero Rectification
rectifyScale = 1
rectL, rectR, projMatL, projMatR, Q, roi_L, roi_R = cv.stereoRectify(newCameraMatrixL,
        distL, newCameraMatrixR, distR, grayL.shape[::-1], rot, trans, rectifyScale,(0,0))

stereoMapL = cv.initUndistortRectifyMap(newCameraMatrixL,distL, rectL, projMatL, grayL.shape[::-1], cv.CV_16SC2)
stereoMapR = cv.initUndistortRectifyMap(newCameraMatrixR,distR, rectR, projMatR, grayR.shape[::-1], cv.CV_16SC2)

#— Saving parameters
print("Saving parameters")
cv_file = cv.FileStorage('C:\\Users\\bianc\\TMvision_TmHttp_server_sample_code\\python_example\\stereoMap.xml',
        cv.FILE_STORAGE_WRITE)

cv_file.write('stereoMapL_x', stereoMapL[0])
cv_file.write('stereoMapL_y', stereoMapL[1])
cv_file.write('stereoMapR_x', stereoMapR[0])
cv_file.write('stereoMapR_y', stereoMapR[1])

cv_file.release()

CamSX = np.array([newCameraMatrixL[0][0],newCameraMatrixL[1][1],newCameraMatrixL[0][2],newCameraMatrixL[1][2]])
CamDX = np.array([newCameraMatrixR[0][0],newCameraMatrixR[1][1],newCameraMatrixR[0][2],newCameraMatrixR[1][2]])
data = {'SX': CamSX , 'DX': CamDX}
tableCam = pd.DataFrame( data, index = ['fx','fy','cx','cy'])

tableCam = tabulate(tableCam, headers = ['SX','DX'])

with open('C:\\Users\\bianc\\TMvision_TmHttp_server_sample_code\\python_example\\StereoMatrices.txt','a') as f:
    f.write(tableCam)
    f.close()
```

A.2 Code: HTTP Server

```
from flask import Flask, jsonify, g, Response, request, flash, redirect, url_for, send_from_directory, render_template
from werkzeug.exceptions import HTTPException
from waitress import serve
from PIL import Image
from werkzeug.utils import secure_filename
from matplotlib import pyplot as plt
from tabulate import tabulate

import os
import io
import cv2
import numpy as np
import datetime
import time
import socket
import requests
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

#--- Undistort and rectify images before giving them to the ORB algorithm to process
#--- Use camera parameters PREVIOUSLY evaluated by stereo calibration script
cv_file = cv2.FileStorage()
cv_file.open(r"C:\Users\bianc\TMvision_TmHttp_server_sample_code\python_example\stereoMap.xml", cv2.FILE_STORAGE_READ)

stereoMapL_x = cv_file.getNode('stereoMapL_x').mat()
stereoMapL_y = cv_file.getNode('stereoMapL_y').mat()
stereoMapR_x = cv_file.getNode('stereoMapR_x').mat()
stereoMapR_y = cv_file.getNode('stereoMapR_y').mat()

cv_file.release()


UPLOAD_FOLDER = r"C:\Users\bianc\TMvision_TmHttp_server_sample_code\python_example\upload"
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}
INDEX = r"C:\Users\bianc\TMvision_TmHttp_server_sample_code\python_example\templates\index.html"
TEMPLATE_DIR = os.path.dirname(os.path.abspath(os.path.dirname(INDEX)))
TEMPLATE_DIR = os.path.join(TEMPLATE_DIR, 'templates')

app = Flask(__name__, template_folder=TEMPLATE_DIR)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

HOST_NAME = 'TM␣Vision␣HTTP␣Server'
HOST_PORT = 80

nu = 0.75



# ============================================================ SYSTEM ============================================================
@app.errorhandler(HTTPException)
#Handle an exception that did not have an error handler associated with it, or that was raised from an error handler.
#This always causes a 500 InternalServerError.
def handleException(e):
    '''Return HTTP errors.'''
    TRIMessage(e)
    return e

#@app.errorhandler(400)
#def bad_request(e):
#     return render_template("400.html"), 400

#@app.errorhandler(404)
#def page_not_found(e):
#     return render_template("404.html"), 404

#@app.errorhandler(405)
#def method_not_allowed(e):
#     return render_template("405.html"), 405

#--- Utility functions
def TRIMessage(message):
    print(f'\n[{datetime.datetime.now(datetime.timezone(datetime.timedelta(0))).astimezone().isoformat(timespec="milliseconds")}]␣{message}')

def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

```
def UndistAndRect(img, stereomap_x, stereomap_y):
    frame = cv2.remap(img, stereomap_x, stereomap_y, cv2.INTER_LANCZOS4, cv2.BORDER_CONSTANT, 0)
    return frame


#— GET
#by deafult the app.route expects a get request. Here an index page is put for the user to open on the pc side.
#@app.route('/')
#def index():
#    return render_template('index.html')

@app.route('/api/<string:m_method>', methods=['GET'])
#dummy GET to try the connection over the TM robot side in the vision node settings
def get(m_method):
    # user defined method
    result = dict()

    if m_method == 'status':
        result = {
            "result": "status",
            "message": "im␣ok"
        }
    else:
        result = {
            "result": "fail",
            "message": "wrong␣request"
        }
    return result

#— POST
@app.route('/api/<string:m_method>', methods=['POST'])
def post(m_method):
    #get key/value
    parameters = request.args
    model_id = parameters.get('model_id')
    TRIMessage(f'model_id:␣{model_id}')
    headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}
    #check key/value
    if model_id is None:
        TRIMessage('model_id␣is␣not␣set')
        result={
            "message": "fail",
            "result": "model_id␣required"
        }
        return jsonify(result)

    #— Saving image on pc
    if 'file' not in request.files:
        flash('No␣file␣part')
        return redirect(request.url)

    file = request.files['file']
    # If the user does not select a file, the browser submits an
    # empty file without a filename.
    if file.filename == '':
        flash('No␣selected␣file')
        return redirect(request.url)

    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        name = filename.rsplit('.')
        if model_id == "dx":
            filename = name[0] + "_DX" + "." + name[1]
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            print('File␣saved␣succesfully')

        if model_id == "sx":
            filename = name[0] + "_SX" + "." + name[1]
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            print('File␣saved␣succesfully')


    #— Image processing
    #The algorithm needs to be trained to track the desired object so
    #the training image is placed in the pc and let the algorithm pick it
    trainim = cv2.imread(r"C:\Users\bianc\TMvision_TmHttp_server_sample_code\python_example\images\train_im.jpg")

    #— The query image is picked from the robot camera and corrected
    if model_id == "dx":
        queryim = cv2.imread(UPLOAD_FOLDER + r"\image_DX.jpg")
        queryim = UndistAndRect(queryim, stereoMapR_x, stereoMapR_y)
    if model_id == "sx":
        queryim = cv2.imread(UPLOAD_FOLDER + r"\image_SX.jpg")
        queryim = UndistAndRect(queryim, stereoMapL_x, stereoMapL_y)


    #— ORB
    #IMPORTANT_NOTE: avoid irrelevant corners in query pictures at all costs!!
    outputimg = queryim.copy()
    width, height = Image.open(UPLOAD_FOLDER + r"\image_DX.jpg").size #widthxheight pixels of query img

    # OpenCV uses BGR as its default colour order for images, matplotlib uses RGB.
    # So if plt is used then uncomment this conversion
    # trainim = cv2.cvtColor(trainim, cv2.COLOR_BGR2RGB)
    # queryim = cv2.cvtColor(queryim, cv2.COLOR_BGR2RGB)

    # Initiate ORB detector
    orb = cv2.ORB_create()

    # find the keypoints and descriptors with ORB
    kp1 = orb.detect(trainim,None)
    kp2 = orb.detect(queryim,None)

    kp1, des1 = orb.compute(trainim, kp1)
    kp2, des2 = orb.compute(queryim, kp2)

    # create BFMatcher object
    bf = cv2.BFMatcher.create(cv2.NORM_HAMMING, crossCheck=False)

    # Match descriptors, knn method.
    matches = bf.knnMatch(des1,des2,k=2)

    # I can also mask the keypoints by "filtering" only the best
    # ones; a.k.a. the keypoints whose descriptor have low distance

    # Sort them in the order of their distance.
    matches = sorted(matches, key = lambda x: x[:][1].distance)

    good = []
    for m,n in matches:
        if m.distance < nu*n.distance:
            good.append([m])

    Matched = cv2.drawMatchesKnn(trainim,kp1,queryim,kp2,
                                 good,outImg=None,matchColor=(0, 155, 0),singlePointColor=(0, 255, 255),matchesMask=None,flags=0)

    kp3 = [] #creating an empty keypoint object

    for i in range(len(good)):
        a = good[i][0].trainIdx
        idx=kp2[a].pt
        key = cv2.KeyPoint(idx[0],idx[1],1)
        kp3.append(key)


    output_img = cv2.drawKeypoints(outputimg, kp3 ,0,(255,0,0),flags=cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS)

    #— Localize object
    obj = np.empty((len(good),2), dtype = np.float32)
    scene = np.empty((len(good),2), dtype = np.float32)
    output = np.empty((len(good),2), dtype=np.float32)

    for i in range(len(good)):
        obj[i,0] = kp1[good[i][0].queryIdx].pt[0] #coordinata x del keypoint delle train image che corrisponde
        # all'i esimo keypoint nella query image— sfrutto il match object che mappa le corrispondenze tra i due set di kypoints
        # nell' obj metto quindi le coordinate in pixel della train image con corrispondenza alla query
        # nell'oggetto scene faccio l'opposto: salvo le coodinate in pixel della query image che corrispondono ai keypoint della train image
        obj[i,1] = kp1[good[i][0].queryIdx].pt[1]
        scene[i,0] = kp2[good[i][0].trainIdx].pt[0]
        scene[i,1] = kp2[good[i][0].trainIdx].pt[1]
    # kp3 is used to create an output image with training image and scene image one next to the other
    for i in range(len(kp3)):
        output[i,0] = kp3[i].pt[0]
        output[i,1] = kp3[i].pt[1]

    try:
        H, _ = cv2.findHomography(obj,scene,cv2.RANSAC)
        H2, _ = cv2.findHomography(obj, output, cv2.RANSAC)
    except:
        print("Not␣enough␣matches␣or␣zero␣matches␣found!")
        result = {
            "message": "Not␣enough␣matches␣or␣zero␣matches␣found!",
            "result": None
        }
        return result #with this return the server goes back to idle state

    #-- Get the corners from the training image
    obj_corners = np.empty((4,1,2), dtype=np.float32)
    obj_corners[0,0,0] = 0
    obj_corners[0,0,1] = 0

    obj_corners[1,0,0] = trainim.shape[1]
    obj_corners[1,0,1] = 0

    obj_corners[2,0,0] = trainim.shape[1]
    obj_corners[2,0,1] = trainim.shape[0]

    obj_corners[3,0,0] = 0
    obj_corners[3,0,1] = trainim.shape[0]

    #-- Here I get the corners of the train object "mapped" on to the query image
    # coordinates through the homography matrix
    # previously evaluated
    scene_corners = cv2.perspectiveTransform(obj_corners, H)
    output_corners = cv2.perspectiveTransform(obj_corners, H2)

    #-- Draw lines between the corners (the mapped object in the scene - image_2 )
    cv2.line(Matched,
            (int(scene_corners[0,0,0] + trainim.shape[1]), int(scene_corners[0,0,1])),\
            (int(scene_corners[1,0,0] + trainim.shape[1]), int(scene_corners[1,0,1])),
            (0,255,0), 4)
    cv2.line(Matched, (int(scene_corners[1,0,0] + trainim.shape[1]), int(scene_corners[1,0,1])),\
    (int(scene_corners[2,0,0] + trainim.shape[1]), int(scene_corners[2,0,1])), (0,255,0), 4)
    cv2.line(Matched, (int(scene_corners[2,0,0] + trainim.shape[1]), int(scene_corners[2,0,1])),\
    (int(scene_corners[3,0,0] + trainim.shape[1]), int(scene_corners[3,0,1])), (0,255,0), 4)
    cv2.line(Matched, (int(scene_corners[3,0,0] + trainim.shape[1]), int(scene_corners[3,0,1])),\
    (int(scene_corners[0,0,0] + trainim.shape[1]), int(scene_corners[0,0,1])), (0,255,0), 4)

    cv2.line(output_img,
        (int(output_corners[0,0,0] ), int(output_corners[0,0,1])),\
        (int(output_corners[1,0,0] ), int(output_corners[1,0,1])),
        (0,255,0), 4)
    # (coordinata x 0,0,0 , coordinata y 0,0,1); le altre due coordinate dopo il backslash sono x,y del punto successivo
    cv2.line(output_img, (int(output_corners[1,0,0] ), int(output_corners[1,0,1])),\
    (int(output_corners[2,0,0] ), int(output_corners[2,0,1])), (0,255,0), 4)
    cv2.line(output_img, (int(output_corners[2,0,0] ), int(output_corners[2,0,1])),\
    (int(output_corners[3,0,0] ), int(output_corners[3,0,1])), (0,255,0), 4)
    cv2.line(output_img, (int(output_corners[3,0,0] ), int(output_corners[3,0,1])),\
    (int(output_corners[0,0,0]), int(output_corners[0,0,1])), (0,255,0), 4)

    #-- Saving outputimage with square
    #if model_id == "dx":
    #     output_img.save(os.path.join(app.config['UPLOAD_FOLDER'], "outputDX_img.jpg"))
    #else:
    #     output_img.save(os.path.join(app.config['UPLOAD_FOLDER'], "outputSX_img.jpg"))

    #-- Get coordinates, height and width of square box
    cx = (output_corners[0,0,0]+output_corners[1,0,0]+output_corners[2,0,0]+output_corners[3,0,0])/4
    cy = (output_corners[0,0,1]+output_corners[1,0,1]+output_corners[2,0,1]+output_corners[3,0,1])/4
    box_h = np.sqrt(np.square(output_corners[0,0,0]-output_corners[3,0,0])+np.square(output_corners[0,0,1]-output_corners[3,0,1]))
    box_w = np.sqrt(np.square(output_corners[0,0,0]-output_corners[1,0,0])+np.square(output_corners[0,0,1]-output_corners[1,0,1]))

    #-- Get rotation of square box
    theta =- np.arctan2(H2[0,1], H2[0,0])
    theta = np.rad2deg(theta)

    if model_id == 'dx':
        label = 'DX_Image'
    else:
        label = 'SX_Image'



#-- Piling result in json format to send back to TMFlow
    # Classification
    if m_method == 'CLS':

        result = {
            "message": "No␣Classification␣method␣implemented,␣yet"
            }

    # Detection
    elif m_method == "DET":

        result = {
            "message":"success",
            "annotations":[
                {
                    "box_cx": float(str(cx)),
                    "box_cy": float(str(cy)),
                    "box_w": float(str(box_w)),
                    "box_h": float(str(box_h)),
                    "label": str(label),
                    "score": float(str(1.000)),
                    "rotation": float(str(theta))

                }
            ],
            "result": "Image" + filename
        }
        # Storing json in txt file but as a table so that matlab can easily read it
        table = [["label",result["annotations"][0]["label"]],["box_cx", result["annotations"][0]["box_cx"]],
            ["box_cy",result["annotations"][0]["box_cy"]],["box_w",result["annotations"][0]["box_w"]],
            ["box_h",result["annotations"][0]["box_h"]],["rotation", result["annotations"][0]["rotation"]]]

        title = "label␣values"
        if model_id == 'dx':
            with open('C:\\Users\\bianc\\TMvision_TmHttp_server_sample_code\\python_example\\jsondx.txt', 'a') as f:
                f.write('\n')
                f.write(str(title))
                f.write('\n')
                f.write(tabulate(table))
                f.close()
        if model_id == "sx":
            with open('C:\\Users\\bianc\\TMvision_TmHttp_server_sample_code\\python_example\\jsonsx.txt', 'a') as f:
                f.write('\n')
                f.write(str(title))
                f.write('\n')
                f.write(tabulate(table))
                f.close()
        print("json␣sent!")

    # no method
    else:
        result = {
            "message": "no␣method",
            "result": None
        }
        with open('json.txt', 'a') as f:
            f.write('\n')
            f.write((str(result)))
            f.close()

    return jsonify(result)


#— Entry point
if __name__ == '__main__':
    check=False
    try:
        host_addr = ([ip for ip in socket.gethostbyname_ex(socket.gethostname())[2] if not ip.startswith("127.")] or
                [[(s.connect(("8.8.8.8", 53)), s.getsockname()[0], s.close()) for s in [socket.socket(socket.AF_INET, socket.SOCK_DGRAM)]][0][1]])
        check = True if len(host_addr) > 0 else False
    except Exception as e:
        TRIMessage(e)
    if check == True:
        host_addr = host_addr[-1] if len(host_addr) > 1 else host_addr[0]
        TRIMessage(f'serving␣on␣http://{host_addr}:{HOST_PORT}')
    else:
        TRIMessage(f'serving␣on␣http://127.0.0.1:{HOST_PORT}')
    serve(app, port=HOST_PORT, ident=HOST_NAME, _quiet=True)
```

A.3 Code: Matlab Class

```
classdef TM<handle
% classe per le funzioni di calcolo terne
    properties
        connessione % tcpip del robot
        ip
        portControl
        vid            % telecamera
        calib          % matrice calibrazione robot
        t              % timer
        G              % Gaussiane per interpolazione
        GL             % Gaussiane ingrandite
        X
        Y
        sended
    end

    methods
        function obj=TM(ip,portControl)
            %Costruttore
            obj.ip = ip;
            obj.portControl = portControl;

            % Connetto il Techman
            obj.connessione = tcpclient(obj.ip,obj.portControl);
            obj.connessione.UserData = [];
            configureTerminator(obj.connessione,"CR/LF");
            obj.connessione.InputBufferSize=50000;
            obj.connessione.OutputBufferSize=50000;

            % Attivo la connessione
            while obj.connessione.NumBytesAvailable == 0
                pause(0.001)
            end
            risp = char(read(obj.connessione,obj.connessione.NumBytesAvailable));
            fprintf('Connessione Control attiva - %s\n',risp(1:end-2))

        end

        function stop(obj)
            comando = '1,StopAndClearBuffer()';
            pack = writepack(obj,"TMSCT",comando);
            write(obj.connessione,pack,"string");
            pause(0.05);
            answer = char(read(obj.connessione,obj.connessione.NumBytesAvailable));

        end

        function answer = disconnect(obj)
            comando = '1,ScriptExit()';
            pack = writepack(obj,"TMSCT",char(comando));
            write(obj.connessione,pack,"string");
            pause(0.05);
            answer = char(read(obj.connessione,obj.connessione.NumBytesAvailable));
            clear obj.connessione
        end

        function startTimer(obj,f,timerCallback,timerCallback_args)
            % Avvia un timer alla frequenza f
            obj.t = timer;

            % Definisco la callback e la modalità
            obj.t.TimerFcn = {timerCallback, timerCallback_args};
            obj.t.ExecutionMode = 'fixedRate';

            % Periodo di esecuzione del timer
            obj.t.Period = 1/f;

            % Salvo il dato
            obj.t.UserData.f = f;
            obj.t.UserData.t = [];

            % Avvio timer
            start(obj.t);
        end

        function cs = checksum(obj,String)
```

```
% Funzione CheckSum
% il checksum è un insieme di due cifre esadecimali che, vengono inviate
% assieme al pacchetto, il robot ricalcola il checkum per conto suo e se
% quello ricalcolato è uguale a quello ricevuto significa che il pacchetto
% è arrivato tutto senza errori.

%function checksum = checksum(String)
% si usa uno XOR esadecimale


String_d = double(String).';     % da char a double numerico, trasponendo
[N,M]=size(String_d); % misuro la dimensione della stringa, in modo da capire
                      % quante volte ripetere l'operazione XOR
cs = String_d(1,:);  %primo valore del CS, ovvero primo valore della String

    for i = 2:N   %ripeto da 2 a N

                cs(1,:) = bitxor(cs(:),String_d(i,:));

    end

cs=dec2hex(cs);%ritrasformo in esadecimale

    if length(cs) == 1  % se il CS risulta ad una sola cifra, ci aggiungo uno zero davanti

        cs = strcat('0',cs);

    end
end

function pack= writepack(obj,head,comando)
    % Funzione writepack
    % Preso il comando, scrive il pacchetto completo da spedire poi al Robot

    %TO UPDATE: rimuovere i write dai comandi di movimento e includerlo in
    %questo metodo

    % comando è il char array che contiene il comando del robot
    % aggiungo Header, che è TMSTA O TMSCT  a seconda di quello che mi serve,
    % quasi sempre TMSCT
    % ottengo la stringa che poi viene analizzata per il checksum, da dollaro ad
    % asterisco eslcusi
    unchecked=char(strcat(head,',',string(length(comando)),',',comando,','));

    %calcolo il checksum
    %cs=TM.checksum(unchecked);
    cs=checksum(obj,unchecked);
    %completo il pacchetto
    pack=strcat('$',unchecked,'*',cs);
    packdouble = [double(pack) 13 10]; % aggiungo il terminator
    packf = char(packdouble); %ritraduco in char e poi in stringa
    pack = packf;
end

function answer = PTP(obj,txt,coord,vel,Ta,racc,PrecPositioning,conf )
    %% PTP point 2 point
    %{


    Effettuo il movimento Point 2 Point dal punto attuale a quello di arrivo.
    per info guardare il manuale Expression-Editor-and-Listen-Node del TM.

    id,PTP(0,1,2,3,4,5,6,7)

    0:Nome della connessione

    1: una string che definisce il formato: 3 lettere
          1: Motion target format:""
           J expressed in joint angles""
           C expressed in Cartesian coordinate
          2: Speed format:""
           P expressed as a percentage
          3: Blending format""
           P expressed as a percentage

    2: coordinate, di giunto in ordine J 1-2-3-4-5-6 in gradi
       oppure cartesiane assolute: x,y,z in mm e Rx Ry Rz
    3: velocità in %
    le velocità scalano con quella che è la % di Progetto Listen
    4: tempo per accelerare a vmax ( Ta) in ms
    5: unione traiettorie ( in % )
    6: Disable precise positioning ( boolean )
                1:true
                    Disable precise positioning
                2:false
                    Enable precise positioning

    7: se uso coordinate Cartesiane specifico la config del robot
       in un vettore {0—1,2—3,4—5}


            0:Braccio destro
            1 Braccio Sinistro

            2: Gomito Alto
            3: Gomito Basso

            4: polso NON flippato
            5: polso Flippato
    %}
    %esempio
    coord = compose("%.4f", coord);
    %esprimo le coordinate contenute in 'coord' come string e le salvo nella
    %stringa 'coo'
    coo=strcat('{',string(coord(1)),',',string(coord(2)),',',string(coord(3)),',',string(coord(4)),',',string(coord(5)),',',string(coord(6)),'}');

    %se nargin>7 ho anche la configurazione e devo adattarla dal file config a
    % un vettore come descritto sopra.
    if nargin >7

    if conf.righty == 1
        a=0;
    else
        a=1;
    end

    if conf.below == 1
        b=3;
    else
        b=2;
    end

    if conf.flip == 1
        c=5;
    else
        c=4;
    end

    % config diventa un file string espresso come '{0-1,2-3,4-5}'
    config=strcat('{',string(a),',',string(b),',',string(c),'}');

    % scrivo il comando, deve risultare in char. La funzione strcat() richiede
    % String o Char, per questo trasformo tutto in String
    comando=char(strcat('1,PTP("',txt,'",',char(coo),',',string(vel),',',string(Ta),',',string(racc),',',string(PrecPositioning),',',config,')'));

    % esempio comando
    %comando='1,PTP("CPP",{400,300,300,180,0,180},10,500,50,false,{1,2,4})';



    else  % caso con nargin =7


    comando=char(strcat('1,PTP("',txt,'",',char(coo),',',string(vel),',',string(Ta),',',string(racc),',',string(PrecPositioning),')'));


    %esempi comando
    %comando='1,PTP("JPP",{61.7399,286.7598,-106.7598,90.0000,-90.0000,-28.2601},15,500,0,false )';
    %comando='1,PTP("JPP",0,0,0,0,0,0,25,500,10,false)'% anche va bene
    %comando='1,PTP("CPP",500,500,600,180,-1,180,10,500,1,true,1,2,4)'
    end

    pack=writepack(obj,"TMSCT",comando);
    write(obj.connessione,pack,"string");
    pause(0.05);
    answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable)); %messaggio di ok
end

function answer=Circle(obj,txt,puntomezzo,puntofin,speed,Ta,racc,ArcAngle,PrecPos)

    %% Movimento CIRCLE
    %{

    id,Cricle(1,2,3,4,5,6,7,8)
    0:nome connesssione
    1: una string che definisce il formato: 3 lettere
           1: Motion target format:""
            C expressed in Cartesian coordinate
           2: Speed format:""
            P expressed as a percentage
                le velocità scalano con quella che è la % di Progetto Listen
            "A" expressed in velocity (mm/s)
           3: Blending format""
            P expressed as a percentage
    2: coordinate cartesiane assolute di un punto dell'arco: x,y,x in mm e Rx Ry Rz
    3: coordinate cartesiane assolute del punto finale: x,y,x in mm e Rx Ry Rz
    4: velocità in % o in mm/s
    5: tempo per accelerare a vmax ( Ta) in ms
    6: unione traiettorie ( in % )
    7:Arc angle(°°), If non-zero value is given, the TCP will keep the same pose and move from
      current point to the assigned arc angle via the given point and end point on arc;
      If zero is given, the TCP will move from current point and pose to end point
      and pose via the point on arc with linear interpolation on pose.
    8:Disable precise positioning ( boolean )
                1:true
                    Disable precise positioning
                2:false
                    Enable precise positioning

    %}
    % i punti devono essere espressi nella forma {1,2,3,4,5,6}, in char, in
    % modo che si possano contare le lettere
    puntomezzo = compose("%.4f", puntomezzo);
    puntofin = compose("%.4f", puntofin);

    pm=strcat('{',string(puntomezzo(1)),',',string(puntomezzo(2)),',',string(puntomezzo(3)),',',string(puntomezzo(4)),',',string(puntomezzo(5)),',',string(puntome
    pf=strcat('{',string(puntofin(1)),',',string(puntofin(2)),',',string(puntofin(3)),',',string(puntofin(4)),',',string(puntofin(5)),',',string(puntofin(6)),'}')

    % scrivo il comando, deve risultare in char. La funzione strcat() richiede
    % String o Char, per questo trasformo tutto in String
    comando=char(strcat('1,Circle("',txt,'",',char(pm),',',char(pf),',',string(speed),',',string(Ta),',',string(racc),',',string(ArcAngle),',',string(PrecPos),')'

    %esempio di comando
    %comando='1,Circle("CAP",{300,300,400,180,0,180},{200,400,400,180,0,180},25,500,10,0,false)';

    pack=writepack(obj,"TMSCT",comando);%scrivo pacchetto

    write(obj.connessione,pack,'string');%invio pacchetto
    pause(0.05);
    answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));%risposta di ok
end

function answer=PVT(obj ,ps ,vs ,coord_type , duration)
    % connessione: tcp device object
    % ps: target position (in joint angles or cartesian coordiates)
    % vs: target speed [mm/s] (it wants the speed in x,y,z,rx,ry,rz!!)
    % coord_type : "J" (joints) or "C" (Cartesian)
    % duration : vector with duration of each movement [ms]

    % PVTEnter(_)
        if coord_type == 'J'

            comando1=strcat('1,PVTEnter(0)',char(13));

        elseif coord_type == 'C'

            comando1=strcat('1,PVTEnter(1)',char(13));

        else

            comando1=strcat('1,PVTEnter()',char(13));
            %It is accepted by the robot and it defaults to Joint coordinates

        end
        comando1 = compose(comando1 + "\n");

    % PVTPoint(_)
        % conversione punto e velocità
        %throw if sizes are different! [later]
        row = size(ps,1); %numero di righe = numero di punti da inserire
        punti = [];
        for i =1:row
```

```
            ps_c(i,1) = compose("%.3f, %.3f, %.3f, %.3f, %.3f, %.3f",ps(i,1:6)); %compose returns a 1x1 string
            vs_c(i,1) = compose("%.3f, %.3f, %.3f, %.3f, %.3f, %.3f",vs(i,1:6));
            %per ogni coppia di righe punto-velocità  produco un comando pvtpoint
            punto=char(strcat('PVTPoint(',ps_c(i),',',vs_c(i),',',compose("%.3f",duration(1,i)),')',char(13)));
            punto = compose(punto +"\n");
            punti = strcat(punti , punto);
            %bisogna farlo così altrimenti col comando precedente non lo interpreta come un andare a capo come vorrebbe invece il robot

        end
    % PVTExit
        comando2='PVTExit()';
        % important note: the last command in a packet of commands must not have the terminator.
        % It has to be added to the whole packet and to all the
        % other commands individually.

    % Final package

        comando = char(compose(comando1 + punti + comando2));
        pack = writepack(obj,"TMSCT",comando);
        write(obj.connessione,pack,"string");
        pause(0.05);
        answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));
end

function q = ask(obj,id)

    %{
    Vado a Chiedere informazioni al TM sul suo stato
    connessione Indica La variabile connessione che sto usando per connettermi
    al TM attraverso TCP-ip

    info indica invece cosa vado a chiedere: in questo informazioni sul
    robot[0], per più info leggere il capitolo 6 del manuale Expression Editor
    and Listen Node

    1=Coordinate Joints da 1 a 6 in gradi[J1,J2,J3,J4,J5,J6]
    2=Coordinate TCP in mm e gradi [x,y,z,rx,ry,rz]
    3= Forza del TCP rispetto alla base
    4= Velocità assoluta del TCP in mm/s

    %}
    %% callbackFcn
    %Il terminator CR\LF è presente alla fine di ogni pacchetto
    %La callback function dcallback viene triggerata ad ogni lettura di CR\LF
    %connessione.BytesAvailableFcnMode = 'terminator';
    %connessione.BytesAvailableFcn = @dcallback;
    configureTerminator(obj.connessione,"CR/LF");
    configureCallback(obj.connessione,"terminator",@dcallback);
    CallbackRunning = []  ;
    d=[];



    %% caso 1: Coordinate Joints
    switch id
        case 1

            comando='1,ListenSend(90,Robot[0].Joint)';% pacchetto da inviare al TM
            pack=writepack(obj,"TMSCT",comando);%funzione che scrive il pacchetto totale
            % da writepack ricevo un array di char
            write(obj.connessione,pack,"string") %invio pacchetto; (implicit cast)
            pause(0.05);
            while not(obj.connessione.UserData ~= double("ok"))
                pause(0.001);
            end
            %- - - - - - - - - - - - - - - - - - - - - - - - - -
            %msg=d((end-29):(end-6)); %tolgo ,*CS dalla fine e conto 24 caratteri,
            % 4 caratteri per valore sono le coordinate in single ( 8bit*4=32 bit))
            % k = find(d == ',',4,'last');
            % msg = d(k(3)+1:k(4)-1);
            % A=double(msg); % trasformo in double i vari caratteri
            % %estraggo i valori dei Ji: ogni 4 caratteri (4*8=32bit) sono il Valore in
            %Single
            %- - - - - - - - - - - - - - - - - - - - - - - - - -
            A = read_message(obj,d);
            %- - - - - - - - - - - - - - - - - - - - - - - - - -
            % J1=single((typecast(uint8(A(1,1:4)), 'single')));
            % J2=single((typecast(uint8(A(1,5:8)), 'single')));
            % J3=single((typecast(uint8(A(1,9:12)), 'single')));
            % J4=single((typecast(uint8(A(1,13:16)), 'single')));
            % J5=single((typecast(uint8(A(1,17:20)), 'single')));
            % J6=single((typecast(uint8(A(1,21:24)), 'single')));
            %
            % q=[ J1,J2,J3, J4, J5, J6];%unisco i Ji nel vettore q
            %- - - - - - - - - - - - - - - - - - - - - - - - - - -
            q = ConvertCoord(obj,A,'joint');

    %% caso 2: Coordinate Robot

       case 2

            comando='1,ListenSend(90,Robot[0].CoordRobot)';% pacchetto da inviare al TM
            pack=writepack(obj,'TMSCT',comando);%funzione che scrive il pacchetto totale
            write(obj.connessione,pack,"string"); %invio il pacchetto
            pause(0.05);
            while not(obj.connessione.UserData ~= double("ok"))
                pause(0.001)
            end
            %- - - - - - - - - - - - - - - - - - - - - - - - - - -
            % k = find(d == ',',3);
            % asterisk = find(d == '*',1);
            % A = d(k(3)+1:asterisk-2);

            %msg=d((end-29):(end-6)); %tolgo ,*CS dalla fine e conto 24 caratteri,
            % 4 caratteri per valore sono le coordinate in single ( 8bit*4=32 bit))
            % k = find(d == ',',4,'last');
            % msg = d(k(3)+1:k(4)-1);
            % A=double(msg); % trasformo in double i vari caratteri
            %estraggo i valori delle coordinate: ogni 4 caratteri (4*8=32bit) sono il
            %Valore in Single
            %- - - - - - - - - - - - - - - - - - - - - - - - - - -
            A = read_message(obj,d);
            % x=single((typecast(uint8(A(1,1:4)), 'single')));
            % y=single((typecast(uint8(A(1,5:8)), 'single')));
            % z=single((typecast(uint8(A(1,9:12)), 'single')));
            % rx=single((typecast(uint8(A(1,13:16)), 'single')));
            % ry=single((typecast(uint8(A(1,17:20)), 'single')));
            % rz=single((typecast(uint8(A(1,21:24)), 'single')));
            %
            % q=[ x,y,z, rx, ry, rz];% unisco le singole coordinate
            q = ConvertCoord(obj,A,'cartesian');
    %% caso 3: TCPForce3D
        case 3

            comando='1,ListenSend(90,Robot[0].TCPForce3D)';% pacchetto da inviare al TM
            pack=writepack(obj,'TMSCT',comando);%funzione che scrive il pacchetto totale
            write(obj.connessione,pack,"string")%invio il pacchetto
            pause(0.05);
            while not(obj.connessione.UserData ~= double("ok"))
                pause(0.001)
            end

            % msg=d((end-9):(end-6)); %tolgo ,*CS dalla fine e conto 4 caratteri,
            % % 4 caratteri per valore sono le coordinate in single ( 8bit*4=32 bit))
            % A=double(msg); % trasformo in double i vari caratteri
            A = read_message(obj,d);
            F=single((typecast(uint8(A(1,1:4)), 'single')));%da double a Single
            q= F;



    %% caso 4: TCPSpeed3D
        case 4

            comando='1,ListenSend(90,Robot[0].TCPSpeed3D)';% pacchetto da inviare al TM
            pack=writepack(obj,'TMSCT',comando);%funzione che scrive il pacchetto totale
            write(obj.connessione,pack,"string");%invio il pacchetto
            pause(0.05);
            while not(obj.connessione.UserData ~= double("ok"))

                pause(0.001)

            end

            % msg=d((end-9):(end-6)); %tolgo ,*CS dalla fine e conto 4 caratteri,
            % % 4 caratteri per valore sono le coordinate in single ( 8bit*4=32 bit))
            % A=double(msg); % trasformo in double i vari caratteri
            A = read_message(obj,d);
            S=single((typecast(uint8(A(1,1:4)), 'single')));%da double a Single
            q=S;
    end


    function dcallback(connessione,event)

            if isempty(CallbackRunning)

                CallbackRunning = true;
                pause(0.05);
                data=char(read(obj.connessione,obj.connessione.NumBytesAvailable));
                pause(0.05);

                datad = double(data);
                % ind = find(datad == 13, 1, 'first');
                % pause(0.05);
                %d = data(1:ind-5);

                msg = char(data);
                ind = find(data == ',',3,'first');
                %estraggo informazione utile
                d = msg(ind(3)+1:end);

                % qui elimino l'ultimo separatore e il checksum
                % dal messaggio
                obj.connessione.UserData = double("ok");
                okay = data(ind:end);
                CallbackRunning = false;

            else

                obj.connessione.UserData = double("ok");

            end

        end

end

function answer = Move_Line(obj,txt,coord,vel,Ta,racc,PrecPos)
    % function answer = Move_Line(connessione,25,500,10,false)
    % function answer = Move_Line([-50,0,0,0,0,0],25,500,10,false)
    %% movimento relativo Line
    %Effettuo il movimento RELATIVO Line dal punto attuale a quello di arrivo.
    %per info guardare il manuale Expression-Editor-and-Listen-Node del TM.
    %{

    id,Move_Line(1,2,3,4,5,6)
    0: nome connessione
    1: una string che definisce il formato: 3 lettere
            1: Motion target format:
             "C": expressed w.r.t current base coordinate""
             T: expressed w.r.t. tool coordinate
            2: Speed format:""
             P expressed as a percentage
             "A" expressed in velocity (mm/s)
            3: Blending format""
             P expressed as a percentage
             "R" expressed as radius
    2: coordinate cartesiane relative alla base: x,y,x in mm e Rx Ry Rz
       coordinate relative al tool (single type strictly needed!!)
    3: velocità in % o in mm/s
    4: tempo per accelerare a vmax ( Ta) in ms
    5: unione traiettorie ( in % ) o in raggio (mm)
    6: Disable precise positioning ( boolean )
                1:true
                    Disable precise positioning
                2:false
                    Enable precise positioning

    %}
    %esprimo le coordinate contenute in 'coord' come string e le salvo nella
    %stringa 'coo'
    coord = compose("%.4f", coord);
    %The TM flow command strictly needs floating point numbers; if inserting a
    %vector of intgers the command will not be executed from robot
    coo=strcat('{',string(coord(1)),',',string(coord(2)),',',string(coord(3)),',',string(coord(4)),',',string(coord(5)),',',string(coord(6)),'}');

    % scrivo il comando, deve risultare in char. La funzione strcat() richiede
    % String o Char, per questo trasformo tutto in String
    comando=char(strcat('1,Move_Line("',txt,'",',char(coo),',',string(vel),',',string(Ta),',',string(racc),',',string(PrecPos),')'));
    %esempio
    %comando='1,Move_Line("CPP",{-50,0,0,0,0,0},25,500,10,false)';
    pack=writepack(obj,"TMSCT",comando);

    write(obj.connessione,pack,"string");
    pause(0.05);
    answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));
end

function answer=Move_PLine(obj,txt,coord,vel,Ta,racc)

    % movimento   Move_Pline
    %{
    %Effettuo il movimento RELATIVO PLine dal punto attuale a quello di arrivo.
    %per info guardare il manuale Expression-Editor-and-Listen-Node del TM.

    id,Move_PLine(1,2,3,4,5,6)

    0:nome connessione
    1: una string che definisce il formato: 3 lettere
           1: Motion target format:
            "C": expressed w.r.t current base coordinate""
            T: expressed w.r.t. tool coordinate""
            J: expressed in joint angles
           2: Speed format:

      ! anche se uso P mi da ok ma non si muove ? mode not present in the manual
            "A" expressed in velocity (mm/s)
           3: Blending format""
            P expressed as a percentage
         ! anche se uso R mi da ok ma non si muove ? mode not present in the
         manual
         motion command parameter includes: "CAP", "TAP", "JAP"


    2: coordinate cartesiane relative alla base: x,y,x in mm e Rx Ry Rz
       coordinate relative al tool
    3: velocità in % o in mm/s
    4: tempo per accelerare a vmax ( Ta) in ms
    5: unione traiettorie ( in % ) o in raggio (mm)

    %}

    coord = compose("%.4f", coord);
    %The TM flow command strictly needs floating point numbers; if inserting a
    %vector of intgers the command will not be executed from robot

    %esprimo le coordinate contenute in 'coord' come string e le salvo nella
    %stringa 'coo'
    coo=strcat('{',string(coord(1)),',',string(coord(2)),',',string(coord(3)),',',string(coord(4)),',',string(coord(5)),',',string(coord(6)),'}');

    % scrivo il comando, deve risultare in char. La funzione strcat() richiede
    % String o Char, per questo trasformo tutto in String
    comando=char(strcat('1,Move_PLine("',txt,'",',char(coo),',',string(vel),',',string(Ta),',',string(racc),')'));

    %Esempio Comando
    %comando='1,Move_PLine("TAP",{00,0,40,0,0,0},100,100,0)';

    pack=writepack(obj,"TMSCT",comando);

    write(obj.connessione,pack,"string");
    pause(0.05);
    answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));
end

function answer = Move_PTP(obj,txt,coord,vel,Ta,racc,PrecPos)
    % move_PTP point 2 point
    %Effettuo il movimento RELATIVO Point 2 Point dal punto attuale a quello di arrivo.
    %per info guardare il manuale Expression-Editor-and-Listen-Node del TM.

    %{
    id,PTP(1,2,3,4,5,6)
    0:nome connessione
    1: una string che definisce il formato: 3 lettere
           1: Motion target format:""
            C: expressed w.r.t. current base
            "T": expressed w.r.t. tool coordinate
            "J": expressed in joint angles note: it is intended as displacement
            of joints NOT reach this set of angles!!

           2: Speed format:""
```

```
             P expressed as a percentage
            3: Blending format""
             P expressed as a percentage

    2: coordinate, di giunto in ordine J 1—2—3—4—5—6 in gradi
       oppure cartesiane assolute: x,y,x in mm e Rx Ry Rz
    3: velocità in %
    le velocità scalano con quella che è la % di Progetto Listen
    4: tempo per accelerare a vmax ( Ta) in ms
    5: unione traiettorie ( in % )
    6: Disable precise positioning ( boolean )
                1:true
                    Disable precise positioning
                2:false
                    Enable precise positioning

    7: conf: se uso coordinate Cartesiane specifico la config del robot
       in un vettore {0—1,2—3,4—5} ???


             0:Braccio destro
             1 Braccio Sinistro

             2: Gomito Alto
             3: Gomito Basso

             4: polso NON flippato
             5: polso Flippato
    %}
    %esempio
    coord = compose("%.3f", coord);
    %The TM flow command strictly needs floating point numbers; if inserting a
    %vector of intgers the command will not be executed from robot

    %solo
    %esprimo le coordinate contenute in 'coord' come string e le salvo nella
    %stringa 'coo'
    coo=strcat('{',string(coord(1)),',',string(coord(2)),',',string(coord(3)),',',string(coord(4)),',',string(coord(5)),',',string(coord(6)),'}');

    % scrivo il comando, deve risultare in char. La funzione strcat() richiede
    % String o Char, per questo trasformo tutto in String
    comando=char(strcat('1,Move_PTP("',txt,'",',char(coo),',',string(vel),',',string(Ta),',',string(racc),',',string(PrecPos),')'));

    pack=writepack(obj,"TMSCT",comando);

    write(obj.connessione,pack,"string");
    pause(0.05);
    answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));
end

function answer=PLine(obj,txt,coord,vel,Ta,racc)
    % movimento Pline
    %{

    Effettuo il movimento PLine dal punto attuale a quello di arrivo.
    per info guardare il manuale Expression—Editor—and—Listen—Node del TM.

    id,PLine(0,1,2,3,4,5)
    0:nome della connessione
    1: una string che definisce il formato: 3 lettere
            1: Motion target format:""
             J: expressed in joint angles""
             C expressed in Cartesian coordinate
            2: Speed format:""
             A: expressed in velocity (mm/s)

           "A" expressed in velocity (mm/s)
            3: Blending format""
             P expressed as a percentage

    2: coordinate cartesiane assolute: x,y,x in mm e Rx Ry Rz
       oppure in gradi di J 1—2—3—4—5—6
    3: velocità in mm/s
    4: tempo per accelerare a vmax ( Ta) in ms
    5: unione traiettorie ( in % )



    %}
    %esempio
    coord = compose("%.4f", coord);
    %esprimo le coordinate contenute in 'coord' come string e le salvo nella
    %stringa 'coo'
    coo=strcat('{',string(coord(1)),',',string(coord(2)),',',string(coord(3)),',',string(coord(4)),',',string(coord(5)),',',string(coord(6)),'}');

    % scrivo il comando, deve risultare in char. La funzione strcat() richiede
    % String o Char, per questo trasformo tutto in String
    comando=char(strcat('1,PLine(”',txt,'”',',char(coo),',',string(vel),',',string(Ta),',',string(racc),')'));

    %esempi
    %comando='1,PLine(”JAP”,{0,0,0,0,0,0},5,500,0)';
    %comando='1,PLine(”CAP”,{500,350,500,180,0,180},25,500,0)';

    pack=writepack(obj,”TMSCT”,comando);

    write(obj.connessione,pack,”string”);
    pause(0.05);
    answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));
end

function answer=Line(obj,txt,coord,speed,Ta,racc,PrecPos)
    % movimento line
    %{

    Effettuo il movimento Line dal punto attuale a quello di arrivo.
    per info guardare il manuale Expression-Editor-and-Listen-Node del TM.


    id,Line(0,1,2,3,4,5,6)
    0:nome connessione
    1: una string che definisce il formato: 3 lettere
           1: Motion target format:””
            C expressed in Cartesian coordinate
           2: Speed format:””
            P expressed as a percentage
            ”A” expressed in velocity (mm/s)
           le velocità scalano con quella che è la % di Progetto Listen
           3: Blending format””
            P expressed as a percentage
            ”R” expressed as radius
    2: coordinate cartesiane assolute: x,y,x in mm e Rx Ry Rz
    3: velocità in % o in mm/s
    4: tempo per accelerare a vmax ( Ta) in ms
    5: unione traiettorie ( in % ) o in raggio (mm)
    6: Disable precise positioning ( boolean )
                1:true
                    Disable precise positioning
                2:false
                    Enable precise positioning


    %}
    coord = compose(”%.4f”, coord);
    %esprimo le coordinate contenute in 'coord' come string e le salvo nella
    %stringa 'coo'
    coo=strcat('{',string(coord(1)),',',string(coord(2)),',',string(coord(3)),',',string(coord(4)),',',string(coord(5)),',',string(coord(6)),'}');

    % scrivo il comando, deve risultare in char. La funzione strcat() richiede
    % String o Char, per questo trasformo tutto in String
    comando=char(strcat('1,Line(”',txt,'”',',char(coo),',',string(speed),',',string(Ta),',',string(racc),',',string(PrecPos),')'));

    %esempio comando
    %comando='1,Line(”CAR”,{400,400,400,180,0,180},10,100,10,false)';

    pack=writepack(obj,”TMSCT”,comando);

    write(obj.connessione,pack,”string”);
    pause(0.05);
    answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));
end

function I=ReadIO(obj,fonte,tipo,id)
    %la funzione di ReadIO restituisce  il valore richiesto con
    %il comando che viene messo in lista d'attesa all'interno del robot.

    %esempio
    %comando='1,IO[”ControlBox”].DO[1]';

    %connessione è il nome della connessione
    %fonte degli output, di base posso scegliere tra ”ControlBox” oppure
    %”EndModule”
    %tipo è DI / AI, se provo a leggere un input dovrebbe darmi errore
```

```
% id è il numero di input, per i digital 0−15 per gli analog da 0 a 6
%


% scrivo il comando, deve risultare in char. La funzione strcat() richiede
% String o Char, per questo trasformo tutto in String
comando=char(strcat('1,ListenSend(90,IO["',string(fonte),'"].',string(tipo),'[',string(id),'])'));
pack=writepack(obj,"TMSCT",comando);
write(obj.connessione,pack,"string");

pause(0.05);
d=char(read(obj.connessione,obj.connessione.NumBytesAvailable));% Pacchetto contenente il valore di output
pause(0.05);

switch tipo
    case "DI"
        ind = strfind(d,'$');
        if length(ind)<2
            ind2 = strfind(d, 'TA');
            if length(ind2)>0
                d=d;
            else
                print("Reinviare il messaggio")
            end
        else
            d = d(1:ind(2)−1);
        end

        msg=d(13:end−6); %tolgo ,*CS dalla fine e conto 4 caratteri,
        % 4 caratteri per valore sono le info in single ( 8bit*4=32 bit))
        A=double(msg); % trasformo in double i vari caratteri
        I=single(uint8(A(1)));
    case "DO"
        ind = strfind(d,'$');
        if length(ind)<2
            ind2 = strfind(d, 'TA');
            if length(ind2)>0
                d=d;
            else
                print("Reinviare il messaggio")
            end
        else
            d = d(1:ind(2)−1);
        end

        msg=d(13:end−6); %tolgo ,*CS dalla fine e conto 4 caratteri,
        % 4 caratteri per valore sono le info in single ( 8bit*4=32 bit))
        A=double(msg); % trasformo in double i vari caratteri
        I=single(uint8(A(1)));
    case "AI"
        ind = strfind(d,'$');
        if length(ind)<2
            ind2 = strfind(d, 'TA');
            if length(ind2)>0
                d=d;
            else
                print("Reinviare il messaggio")
            end
        else
            d = d(1:ind(2)−1);
        end

        msg=d(13:end−6); %tolgo ,*CS dalla fine e conto 4 caratteri,
        % 4 caratteri per valore sono le info in single ( 8bit*4=32 bit))
        A=double(msg); % trasformo in double i vari caratteri
        I=single((typecast(uint8(A), 'single')));
    case "AO"
        ind = strfind(d,'$');
        if length(ind)<2
            ind2 = strfind(d, 'TA');
            if length(ind2)>0
                d=d;
            else
                print("Reinviare il messaggio")
            end
        else
            d = d(1:ind(2)−1);
        end

        msg=d(13:end−6); %tolgo ,*CS dalla fine e conto 4 caratteri,
        % 4 caratteri per valore sono le info in single ( 8bit*4=32 bit))
```

```
                A=double(msg); % trasformo in double i vari caratteri
                I=single((typecast(uint8(A), 'single')));
        end

end

    function answer=WriteIO(obj,fonte,tipo,id,valore)
        %la funzione di IstantReadIO scrive SUBITO il valore richiesto senza
        %che il comando sia messo in lista d'attesa all'interno del robot.

        %esempio comando
        %comando='1,IO["ControlBox"].DO[1]=1';

        %connessione è il nome della connessione
        %fonte degli output, di base posso scegliere tra "ControlBox" oppure
        %"EndModule"
        %tipo è DO / AO, se provo a scrivere un input dovrebbe darmi errore
        % id è il numero di input, per i digital 0-15 per gli analog da 0 a 6
        %

        % scrivo il comando, deve risultare in char. La funzione strcat() richiede
        % String o Char, per questo trasformo tutto in String

        switch tipo
            case "AI"
                answer = fprintf("Cannot write on input channel. \n");
                return
            case "DI"
                answer = fprintf("Cannot write on input channel. \n");
                return
            otherwise
                comando=char(strcat('1,IO["',string(fonte),'"].Instant',string(tipo),'[',string(id),']=',string(valore)));
                pack=writepack(obj, 'TMSCT',comando);
                write(obj.connessione,pack);
                pause(0.05);
                answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));% Pacchetto contenente il valore di output
        end
    end

    function I=InstantReadIO(obj,fonte,tipo, id)
        % Descrizione
        %la funzione di IstantReadIO restituisce SUBITO il valore richiesto senza
        %che il comando sia messo in lista d'attesa all'interno del robot.

        %esempio di comando che devo comporre
        %comando='1,IO["ControlBox"].DO[1]';

        % INPUT
        %connessione è il nome della connessione
        %fonte degli output, di base posso scegliere tra "ControlBox" oppure
        %"EndModule"
        %tipo è DI / AI, se provo a leggere un input dovrebbe darmi errore
        % id è il numero di input, per i digital 0-15 per gli analog da 0 a 6

        % scrivo il comando, deve risultare in char. La funzione strcat() richiede
        % String o Char, per questo trasformo tutto in String
        comando=char(strcat('1,ListenSend(90,IO["',string(fonte),'"].Instant',string(tipo),'[',string(id),'])'));
        pack=writepack(obj, 'TMSCT',comando);
        write(obj.connessione,pack,"string");

        pause(0.05);
        d=read(obj.connessione,obj.connessione.NumBytesAvailable);% Pacchetto contenente il valore di output
        d = char(d);
        pause(0.05);

        switch tipo
            case "DI"
                ind = strfind(d,'$');
                if length(ind)<2
                    ind2 = strfind(d, 'TA');
                    if length(ind2)>0
                        d=d;
                    else
                        print("Reinviare il messaggio")
                    end
                else
                    d = d(1:ind(2)-1);
                end

                msg=d(13:end-6); %tolgo ,*CS dalla fine e conto 4 caratteri,
                % 4 caratteri per valore sono le info in single ( 8bit*4=32 bit))
            A=double(msg); % trasformo in double i vari caratteri
            I=single(uint8(A(1)));
        case "DO"
            ind = strfind(d,'$');
            if length(ind)<2
                ind2 = strfind(d, 'TA');
                if length(ind2)>0
                    d=d;
                else
                    print("Reinviare il messaggio")
                end
            else
                d = d(1:ind(2)-1);
            end
            %pause(0.05)
            msg=d(13:end-6); %tolgo ,*CS dalla fine e conto 4 caratteri,
            % 4 caratteri per valore sono le info in single ( 8bit*4=32 bit))
            A=double(msg); % trasformo in double i vari caratteri
            I=single(uint8(A(1)));
        case "AI"
            ind = strfind(d,'$');
            if length(ind)<2
                ind2 = strfind(d, 'TA');
                if length(ind2)>0
                    d=d;
                else
                    print("Reinviare il messaggio")
                end
            else
                d = d(1:ind(2)-1);
            end

            msg=d(13:end-6); %tolgo ,*CS dalla fine e conto 4 caratteri,
            % 4 caratteri per valore sono le info in single ( 8bit*4=32 bit))
            A=double(msg); % trasformo in double i vari caratteri
            I=single((typecast(uint8(A), 'single')));
        case "AO"
            ind = strfind(d,'$');
            if length(ind)<2
                ind2 = strfind(d, 'TA');
                if length(ind2)>0
                    d=d;
                else
                    print("Reinviare il messaggio")
                end
            else
                d = d(1:ind(2)-1);
            end

            msg=d(13:end-6); %tolgo ,*CS dalla fine e conto 4 caratteri,
            % 4 caratteri per valore sono le info in single ( 8bit*4=32 bit))
            A=double(msg); % trasformo in double i vari caratteri
            I=single((typecast(uint8(A), 'single')));
    end
end

function answer=InstantWriteIO(obj,fonte,tipo,id,valore)
    %la funzione di IstantReadIO scrive SUBITO il valore richiesto senza
    %che il comando sia messo in lista d'attesa all'interno del robot.

    %esempio comando
    %comando='1,IO["ControlBox"].DO[1]=1';

    %connessione è il nome della connessione
    %fonte degli output, di base posso scegliere tra "ControlBox" oppure
    %"EndModule"
    %tipo è DO / AO, se provo a leggere un input dovrebbe darmi errore
    % id è il numero di input, per i digital 0-15 per gli analog da 0 a 6
    %

    % scrivo il comando, deve risultare in char. La funzione strcat() richiede
    % String o Char, per questo trasformo tutto in String
    switch tipo
        case "AI"
            answer = fprintf("Cannot write on input channel. \n");
            return
        case "DI"
            answer = fprintf("Cannot write on input channel. \n");
            return
        otherwise
            comando=char(strcat('1,IO["',string(fonte),'"].Instant',string(tipo),'[',string(id),']=',string(valore)));
```

```
                pack=writepack(obj, "TMSCT",comando);
                write(obj.connessione,pack);
                pause(0.05);
                answer=char(read(obj.connessione,obj.connessione.NumBytesAvailable));% Pacchetto contenente il valore di output
            end
        end

        function A = read_message(obj,msg)

            % trovo le prime tre virgole che sono sempre presenti e so che dopo di
            % quella fino alla fine del messaggio (per come arriva il messaggio
            % dalla callback di ask) trovo l'informazione utile
            msg = char(msg);
            %k = find(msg == ',',3,'first');
            %prima mi assicuro che non ci sia il messaggio di ok
            numdoll = find(msg == '$');
            if length(numdoll) == 0 || length(numdoll) == 1
                ast = find(msg == '*');
                if length(ast) == 1
                    d = msg(1:ast-2);
                else
                    aste = length(ast);
                    d = msg(1:ast(1,aste-1)-2);
                end
            else
                ast = find(msg == '*');
                if length(ast) == 1
                    d = msg(1:ast-2);
                else
                    aste = length(ast);
                    %l'ultimo dollaro sarà quello del pacchetto ok
                    %WIP
                    msg = msg(1:numdoll(1,length(numdoll)));
                    d = msg(1:ast(1,aste-1)-2);
                end
            end
            %estraggo informazione utile

            %converto messaggio in double e lo restituisco
            A = double(d);

        end

        function q = ConvertCoord(obj,A,type)

            switch type
            case "joint"

                J1=single((typecast(uint8(A(1,1:4)), 'single')));
                J2=single((typecast(uint8(A(1,5:8)), 'single')));
                J3=single((typecast(uint8(A(1,9:12)), 'single')));
                J4=single((typecast(uint8(A(1,13:16)), 'single')));
                J5=single((typecast(uint8(A(1,17:20)), 'single')));
                J6=single((typecast(uint8(A(1,21:24)), 'single')));

                q=[ J1,J2,J3, J4, J5, J6];

            case "cartesian"

                x=single((typecast(uint8(A(1,1:4)), 'single')));
                y=single((typecast(uint8(A(1,5:8)), 'single')));
                z=single((typecast(uint8(A(1,9:12)), 'single')));
                rx=single((typecast(uint8(A(1,13:16)), 'single')));
                ry=single((typecast(uint8(A(1,17:20)), 'single')));
                rz=single((typecast(uint8(A(1,21:24)), 'single')));

                q = [ x,y,z, rx, ry, rz];

            end

        end


    end
    methods (Static)

        answer = FT(connessione,id)
        answer = SendVariables(connessione,variables)

    end
end
```

A.4 Code: Matlab script

```
clear all
close all
clc

%% -- Offline points and Transformation matrices
chess_origin = [102, -475, 0, 179, 0, 0]; % I put chessboard on the table
camera1origin = [38, -304, 400, 179, 0, 0]; %left frame
camera2origin = [-43, -304, 400, 179, 0, 0]; %right frame

% ALL these coordinates are w.r.t. ROBOT FRAME and are the FLANGE POSITIONS
Ty = trans.Y(180);

%-- From world to robot frame
TWR = [1, 0, 0, chess_origin(1); 0, 1, 0, chess_origin(2); 0, 0, 1, chess_origin(3); 0, 0, 0, 1];
TWR = Ty*TWR;

Lty=0.079*1000; %orizzontale (y)
Ltz=0.0522*1000; %verticale (z)
%-- From Camera to robot flange offset
TCaF=trans.Cardano(0,Lty,Ltz,0,0,0);

%% -- Focal length evaluation
work_dist = 300;
% FOV at 300 mm of working distance
horiz_FOV = 281.6;
vert_FOV = 211.2;

img_wid = 2592; %image width in pixel
img_h = 1944;%image height in pixel

% singular pixel size
px_size_w = abs(tan(horiz_FOV/2)*work_dist/img_wid);
px_size_h = abs(tan(vert_FOV/2)*work_dist/img_h);

hor_asp = horiz_FOV/vert_FOV;
vert_asp = vert_FOV/horiz_FOV;

megpx = 5.00*1e6;
a = hor_asp*megpx;
b = vert_asp*megpx;

rad2deg = 180/pi;
deg2rad = pi/180;

res_h = sqrt(a); %resolution of camera
res_v = sqrt(b);

%sensor size
sens_h = px_size_w*res_h; % h for horizontal
sens_v = px_size_h*res_v;
WD = 300;
AFOV = 2*rad2deg*atan(horiz_FOV/(2*WD)); %result is in radians!

%FOCAL LENGTH
%f = fxS * sens_h/img_wid
f = sens_h/(2*(rad2deg*tan((deg2rad*AFOV))/2))

%% -- Reading Stereo Calibration info
%-- Reading Camera Matrices from data (comes from calibration on python)
fid = readtable('C:\Users\bianc\TMvision_TmHttp_server_sample_code\python_example\StereoMatrices.txt','ReadRowNames',true);
% from camera to Image plane
K = [f, 0, 0, 0;
    0, f, 0, 0;
    0, 0, 1, 0];
% from image plane to pixel
K_intr_S = [fid(1,1).Variables, 0, fid(3,1).Variables;
            0, fid(2,1).Variables, fid(4,1).Variables;
            0, 0, 1];
K_intr_D = [fid(1,2).Variables, 0, fid(3,2).Variables;
            0, fid(2,2).Variables, fid(4,2).Variables;
            0, 0, 1];

%-- Intrinsic matrices
KS = K_intr_S*K;
KD = K_intr_D*K;

%% -- json info
```

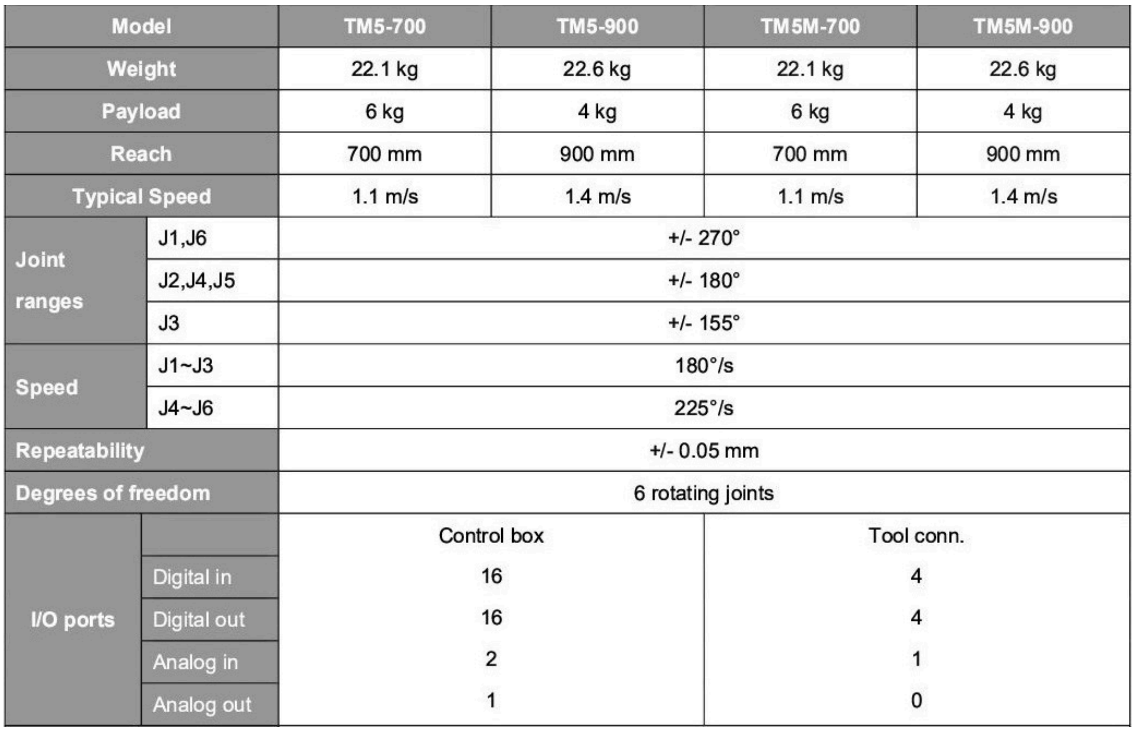
```
% Now I have to use the info in the json in order to identify the pixels
% that correspond to the object border and pick the corresponding 3D
% coordinates in the worldPoints object
json_storagedx = "C:\Users\bianc\TMvision_TmHttp_server_sample_code\python_example\jsondx.txt";
json_storagesx = "C:\Users\bianc\TMvision_TmHttp_server_sample_code\python_example\jsonsx.txt";

jsontabledx = readtable(json_storagedx, 'VariableNamingRule','preserve');
jsontablesx = readtable(json_storagesx, 'VariableNamingRule','preserve');

c_DX = [jsontabledx(1,2).Variables,jsontabledx(2,2).Variables,1];
c_SX = [jsontablesx(1,2).Variables,jsontablesx(2,2).Variables,1];

rot_DX = jsontabledx(5,2).Variables;
rot_SX = jsontablesx(5,2).Variables;

%-- Clearing text files
fopen(json_storagedx,'w');
fopen(json_storagesx,'w');

%% -- Evaluating depth
baseline = 81; % [mm]
%alpha = 1;%1 to use extra FOV near images edge, less than 1 and greater than 0 otherwise
fxD=K_intr_D(1,1);
fxS=K_intr_S(1,1);
S_x = c_SX(1,1);
D_x = c_DX(1,1);
disp = S_x-D_x;
z_S = abs(baseline*fxS)/disp;
z_D = abs(baseline*fxD)/disp;
z = abs((z_D+z_S)/2)

%% -- Building 3D coordinate of interest and pick up routine
% note: rotation on z axis given by json info about rotation!
%from pixel to image
desired_DX_t = inv(K_intr_D(1:3,1:3))*c_DX';
desired_SX_t = inv(K_intr_S(1:3,1:3))*c_SX';

%from image plane of each camera to image plane of object
zp = baseline/(abs(desired_SX_t(1,1))/f+abs(desired_DX_t(1,1))/f);
x = 0.5*zp*(desired_SX_t(1,1)/f-desired_DX_t(1,1)/f);
y = zp*desired_SX_t(2,1)/f;

desired_implane = [x;y;z]
%coordinates of point in world coordinates

des2= TCaF*TWR*[desired_implane;1]; % ok so this worked

% TcWR is from world to robot coordinate
% desired_implane is in world coordinate
% and then the physical offset of the distance between flange and camera is
% added

des2(3,1)=camera1origin(1,3)

%% -- Averaging rotation
if rot_DX<0 && rot_SX<0
    rot_ = -mean([abs(rot_DX),abs(rot_SX)])
else
    rot_ = mean([rot_DX,rot_SX])
end

%-- Keeping relative rotation in between 0 and 90 degrees
if rot_ >= 90
    rot = rot_ -180
elseif rot_ <= -90
    rot = rot_ + 180
else
    rot = rot_
end

%-- Calcolo altezza
base_pinza = 60;
grip_height = 56; %[mm]
half_finger_height = 20; %[mm]
flange_to_camera_height = 48.6; % [mm]
% epsilon = abs(grip_height+half_finger_height-flange_to_camera_height);
epsilon = abs(grip_height+half_finger_height-flange_to_camera_height);
altezza = des2(3,1) - z + epsilon

%% Robot pick up routine
% %-- Connect Robot-----------------------------------------------------
```

```
src =TM('10.10.10.160',5890);
%% -- Move above the evaluated 3D pose ----------------------------------
PLine(src, 'CAP',[des2(1),des2(2),des2(3),179,0,0],200,100,100);
%-- Rotate gripper above the object
Move_PLine(src, 'CAP',[0,0,0,0,0,rot],200,100,100);

%%-- Open gripper ------------------------------------------------------
%% high == chiuso 1
%% low == aperto 0

Read = InstantReadIO(src, 'ControlBox', 'DO',0);
pause(0.05)
if  Read == 1
    WriteIO(src, 'ControlBox', 'DO',0,0);
end
pause(0.05)

%%-- Descend
PLine(src, 'CAP',[des2(1),des2(2),altezza,179,0,rot],200,100,100);

%%-- Close gripper
pause(1)
WriteIO(src, 'ControlBox', 'DO',0,1);

%%-- Move away
Move_PLine(src, 'CAP',[0,0,des2(3),0,0,0],200,100,100);
pause(5)

%%-- Go to cardboard box
PLine(src, 'CAP',[243.44, -105.93, 400, -179,0,90],200,100,100);
pause(5)

%%-- Descend to box
PLine(src, 'CAP',[243.44, -105.93, 230, -179,0,90],200,100,100);
pause(5)

%%-- Let go of object
WriteIO(src, 'ControlBox', 'DO',0,0);
pause(2)

%%-- Move upwards
PLine(src, 'CAP',[243.44, -105.93, 400, -179,0,90],200,100,100);
% pause(5)

%%-- Disconnect Robot
pause(10)
disconnect(src);
```

Appendix B

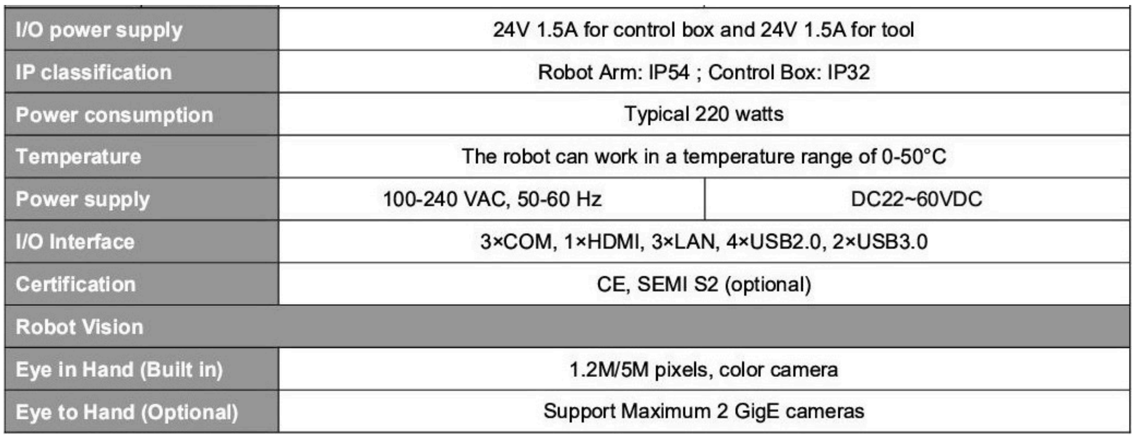
Hardware Specification

B.1 TM5-700 Hardware specification



Model		TM5-700	TM5-900	TM5M-700	TM5M-900
Weight		22.1 kg	22.6 kg	22.1 kg	22.6 kg
Payload		6 kg	4 kg	6 kg	4 kg
Reach		700 mm	900 mm	700 mm	900 mm
Typical Speed		1.1 m/s	1.4 m/s	1.1 m/s	1.4 m/s
Joint ranges	J1,J6	+/- 270°			
	J2,J4,J5	+/- 180°			
	J3	+/- 155°			
Speed	J1~J3	180°/s			
	J4~J6	225°/s			
Repeatability		+/- 0.05 mm			
Degrees of freedom		6 rotating joints			
I/O ports		Control box		Tool conn.	
	Digital in	16		4	
	Digital out	16		4	
	Analog in	2		1	
	Analog out	1		0	

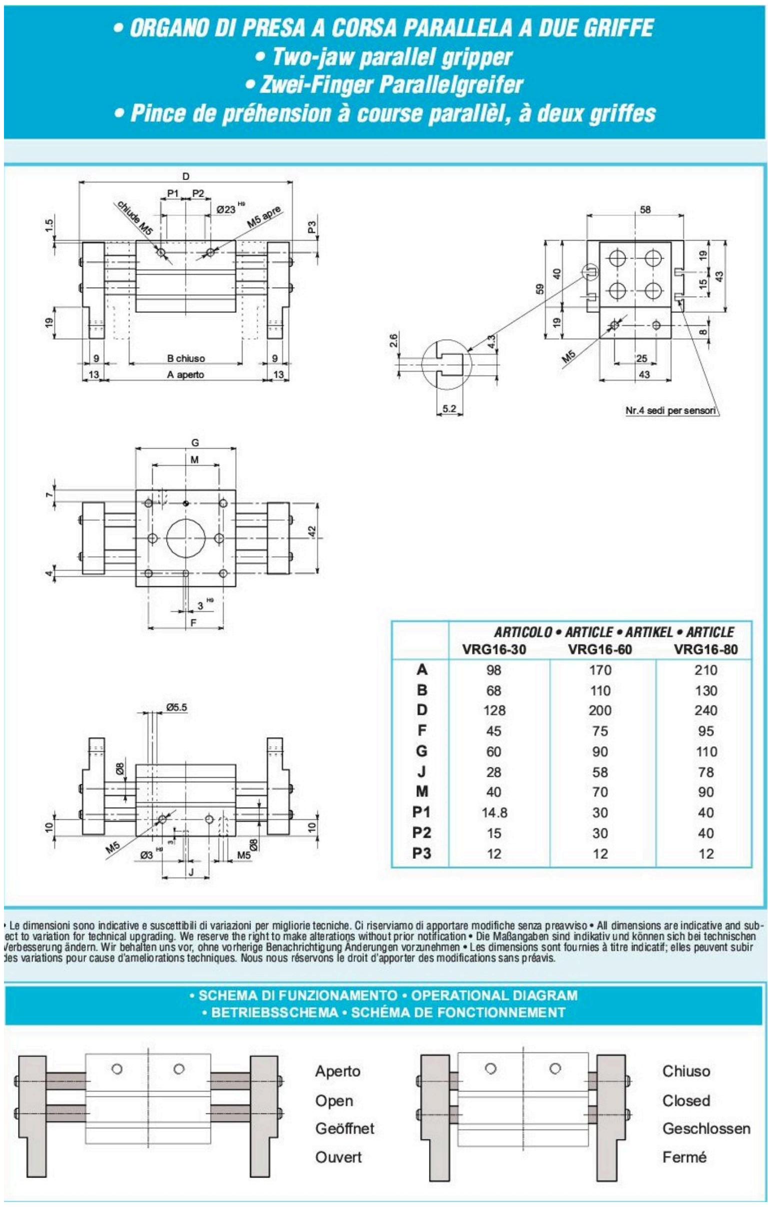
Figure B.1: Datasheet of `TM5-700`



I/O power supply	24V 1.5A for control box and 24V 1.5A for tool	
IP classification	Robot Arm: IP54 ; Control Box: IP32	
Power consumption	Typical 220 watts	
Temperature	The robot can work in a temperature range of 0-50°C	
Power supply	100-240 VAC, 50-60 Hz	DC22~60VDC
I/O Interface	3×COM, 1×HDMI, 3×LAN, 4×USB2.0, 2×USB3.0	
Certification	CE, SEMI S2 (optional)	
Robot Vision		
Eye in Hand (Built in)	1.2M/5M pixels, color camera	
Eye to Hand (Optional)	Support Maximum 2 GigE cameras	

Figure B.2: Datasheet of `TM5-700`

B.2 Gripper specification



• ORGANO DI PRESA A CORSA PARALLELA A DUE GRIFFE
• Two-jaw parallel gripper
• Zwei-Finger Parallelgreifer
• Pince de préhension à course parallèl, à deux griffes

	ARTICOLO • ARTICLE • ARTIKEL • ARTICLE		
	VRG16-30	**VRG16-60**	**VRG16-80**
A	98	170	210
B	68	110	130
D	128	200	240
F	45	75	95
G	60	90	110
J	28	58	78
M	40	70	90
P1	14.8	30	40
P2	15	30	40
P3	12	12	12

• Le dimensioni sono indicative e suscettibili di variazioni per migliorie tecniche. Ci riserviamo di apportare modifiche senza preavviso • All dimensions are indicative and sub-ect to variation for technical upgrading. We reserve the right to make alterations without prior notification • Die Maßangaben sind indikativ und können sich bei technischen Verbesserung ändern. Wir behalten uns vor, ohne vorherige Benachrichtigung Änderungen vorzunehmen • Les dimensions sont fournies à titre indicatif; elles peuvent subir des variations pour cause d'ameliorations techniques. Nous nous réservons le droit d'apporter des modifications sans préavis.

• SCHEMA DI FUNZIONAMENTO • OPERATIONAL DIAGRAM
• BETRIEBSSCHEMA • SCHÉMA DE FONCTIONNEMENT

Figure B.3: Gripper specifications

References

[1] J. Shaw and K. Y. Cheng, “Object identification and 3-d position calculation using eye-in-hand single camera for robot gripper,” in *2016 IEEE International Conference on Industrial Technology (ICIT)*, 2016, pp. 1622–1625.

[2] A. Cherubini and D. Navarro-Alarcon, “Sensor-based control for collaborative robots: Fundamentals, challenges, and opportunities,” *Frontiers in Neurorobotics*, vol. 14, 2021.

[3] Doxygen. “Understanding features.” (), [Online]. Available: `https://docs.opencv.org/3.4/df/d54/tutorial_py_features_meaning.html`. (accessed: 30/09/23).

[4] N. Gregory Hollows. “Understanding focal length and field of view.” (), [Online]. Available: `https://www.edmundoptics.com/knowledge-center/application-notes/imaging/understanding-focal-length-and-field-of-view/`.

[5] Doxygen. “Camera calibration and 3d reconstruction.” (), [Online]. Available: `https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga93efa9b0aa890de240ca32b11253dd4a`. (accessed: 30/09/23).

Acknowledgments

I want to thank the Supervisor who gave me the opportunity for this thesis which was an occasion to learn new skills and apply what I learned through the years.

I also want to thank Riccardo whose valuable tips and suggestions helped a lot in developing this thesis.

I want to properly thank my parents who always cheered on me and believed in me, even when I did not give myself enough credit.

I want to thank my boyfriend who stayed by my side during the majority of the university career and who helped me believe in myself. Thank you for being the sunlight ray that you are, and for making everything fun.

I want to thank all the girls from the second floor of the ”Collegio Sorelle della Misericordia” with whom I shared many joyful days and made many exam sessions much lighter. I will very much miss you all everyday of my life.