POLITECNICO DI TORINO

Master's degree in Mechatronic Engineering

Academic year 2020/2021

July 2021

6DOF Pose estimation for space applications

Politecnico di Torino

Supervisor
Prof. Marcello Chiaberge

Candidate
Giusy D'Amico

Supervisor at Thales Alenia Space
Ing. Andrea Merlo

A mia mamma che mi insegna a non arrendermi mai
A mia zia che è sempre con me

Abstract

This thesis presents the methods used to estimate the pose of an object. The pose estimation can be done with two possible approaches, one is using markers and the second is model based. This work is divided in two parts, the first part presents the markers based method and the second part the model based method. The first method is implemented in Python using OpenCV, the second method is implemented in C++ using OpenCV and VISP,it is a modular software package done by INRIA.

In order to estimate the position and orientation of the object using the markers based method, it is necessary have at least four points, but in order to increase robustness and achieve good results it is better to use at least six markers. In space applications this method can be limiting, because requires that these markers have to be around the target. Even if, this approach has a great accuracy. In this work this method is implemented with three different kind of markers. The first markers Library has been created freely by choosing some symbols. The second and third has been provided respectively by pyzbar and Aruco Library. The detection of markers in the first application has been done through the contours of markers, in the second and third applications using their library. The pose estimation has been done using OpenCV, in particular given the assumption that all markers are in the same plane, it exploits the 2D points to estimate the pose.

The model based method, is more free, because in order to estimate the pose of object does not requires the presence of markers but it is necessary know only the model of the object. In this work is used an circular object with two bolts, the following object has been chosen because it reproduces, on a small scale, a launch adapter ring (LAR) of satellite. It has been used two different cameras, one is the stereo fisheye camera T265 and the second is the depth camera D435, in this way it is possible to see two different methods in order to extract the 3D sets of point clouds. In particular after the first detection of object, the tracking has been done using VISP, and the estimation of pose has been done implemented some algorithms of 3D point cloud registration, like ICP and CPD.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The computer vision is the scientific field that gives to machines the possibility to understand the visual world through digital images. It includes some tasks as acquiring, processing and analyzing digital images, in order to extract information from the real world. Also the object detection, 3D pose estimation are sub-domain of computer vision.
The first application of computer vision dates back to 1966, when a group of students wanted to attach a camera on computer in order to "describe what it saw".[1] [2] In the next years the scientists analyze the mathematical world of computer vision, as the scale-space, the contours model. In the last years the computer vision is a research field constantly evolving.
In nowadays the majority of applications use the power of computer vision from driverless cars to building autonomous robots for satellite missions.
In space applications the computer vision is used in order to automatize some actions, especially during space explorations. For example during Mars exploration the computer vision has a great role to detect obstacles and to avoid them. Another case is the spacecraft docking, the computer vision has the great power to estimate the position and orientation of the target spacecraft in 3-dimensional space. The pose computed by the computer vision system are then used by the guidance and control loops of the docking system to control the spacecraft orientation in six degrees of freedom which is essential during docking.[3]
The tracking and pose estimation of object can be done with two possible approaches **markers based** and **model based** (or markerless). In the first part is described the approaches used for the markers based method, in the second part of the work is presented the model based method.

Part I

First Part

Chapter 2

Markers based

Visual servoing, with markers, is one of the most used technique. These markers are positioned around the object that the camera has to recognise , and the camera has to:

1) **Detect**: it acquires the video and has to detect the markers.

2) **Recognize**: after has detected the markers, it is necessary pre-processing images, in order to extrapolate some features of markers, this is done in order to recognize them.

3) **Estimation of position and orientation**: these markers are used to compute its position and orientation respect the camera.

2.1 State of art

A markers is artificial object that is placed into a scene in order to supply a reference frame. Usually markers are used when high precision are required or when there is a lack of object's model. In fact, for these reasons, markers continue to be used for the object detection and pose estimation. They are generally designed to be easily detected and recognized in images.[4] Usually they have to be asymmetric, and invariant to projection, for example it is not recommended to use ellipse or circle , because these geometrical entities are not invariant to projective transformation. However, there are different works that used circles as markers like in the work proposed by Gatrell [5], using a contrasted concentric circles, he extracted the centroids of these circles that are invariant to the three translational and to one rotation degrees of freedom, the extractions was robust. In the following years, this method was modified with alternating concentric circles white and black color or adding colors and multiple scales [6], presents a light-invariant circular fiducial detection method that uses relations among fiducials and their backgrounds for segmenting regions of an image. In 2005, [7] introduce a set of four circles located at the corner of a square, an identification pattern is placed at the centroid of the four dots in order to distinguish between different

targets. This approach introduced the most used methods of today, and it was really useful because introduced the recognition of fiducial markers in case of complex scene when there are more tags. Today there are a lot of library that use artificial tags one of this is pyzbar, that used the QrCode as markers, they can store alphanumeric characters, other library that provides a visual markers which stores just a simple binary code are ARToolkit, AprilTags and Aruco. In this work has been chosen to use Aruco because it has the fastest performance and the smallest standard deviation (of the compute time per frame) among the three libraries, so it is the best choice for a real-time applications [8]. Finally, it has been designed some markers that are not conventional as Aruco or QrCode, it used a geometrical figure or symbol, it can be seen in *myMarkers* application.

2.2 Implemented methods

The computer vision based on markers has been implemented with three different kind of markers and for all applications it has been used a pinhole camera:

1) **myMarkers**: it has chosen some symbols and they have been used as markers.

2) **QrCode**: it has used the QrCode as markers.

3) **ArucoMarkers**: it is used the Aruco library 5x5.

All of these approaches have some common process, but they have advantages and disadvantages.
In particular, all of these methods following the same steps:

1) **Calibration of camera**.

2) **Pre-processing of image**.

3) **Detection and recognition of markers**.

4) **Estimation of position and orientation of markers respect camera**.

2.3 Camera Calibration

Before starting the processing of images, it is necessary to compute the intrinsic parameters of camera. The intrinsic parameters of camera, are specific of camera, and they are used in order to correct lens distortion and to compute the location of camera. The intrinsic parameters are the focal length (f_x, f_y), optical centers (c_x, c_y). From them it is possible to build the camera matrix, it is unique for the same camera, so it can be used for all images taken from the same camera.

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

In order to correct these distortions, it is important to know the coordinates of these points in real world space.
Using the **OpenCV** library, it is possible to do the calibration of camera taking at least 10 test patterns, the possible test patterns can be the **ChessBoard**, how is shown in the Fig. 2.1 or directly the *ArucoMarkers*.

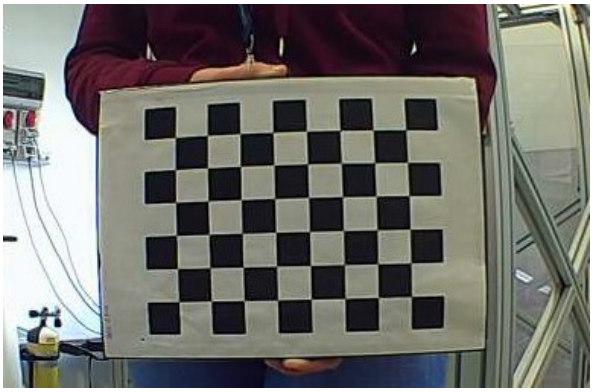


Figure 2.1: Chess Board

2.3.1 Structure

The objective is, starting from the 3D world points, to find the corresponding 2D points. In particular, the 2D points are called **image points**, they are the coordinates of points in the image plane, expressed in pixels.
The 3D world points are called **objective points**, their coordinates **X,Y,Z**, but for simplicity is chosen the points of chess board in XY plane, with Z=0, because all points are on the same plane.

2.3.2 Computation of camera matrix

It has been used a chess board (8,6), considering the internal corners. The objective points have been chosen following the structure (0,0,0), (1,0,0),(2,0,0),(7,5,0), they are

the points from left to right, from top to bottom, and then these points are multiplied with the real dimension of each square in *mm* in order to have more accuracy, in order to give a scale.
Before to compute the image points, it is necessary to process the acquired images. The acquired images are in RGB color space, and it is necessary to convert from RGB to gray color space. It is used the command (from **OpenCV**):

```
cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
```

After, it is used the command to detected coordinates in the image plane, using the command:

```
cv2.findChessBoardCorners(img, (8,6), None, None)
```

In order to improve and determine their positions more accurately is used the function:

```
cv2.cornerSubPix(img,corners, (11,11), (-1,-1), criteria)
```

The term *criteria* defines the termination criteria for iterative algorithms, where chosen the maximum numbers of iterations, and the desired accuracy at which the iterative algorithm stops.

```
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30,
                    0.001)
```

Having the object points and image points it is possible to calibrate the camera:

```
mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img,
                    None,None)
```

Where as input:

- **object Points**: are the 3D points of the chess board,with Z=0.
- **imgpoints**: are the 2D points on the image plane.
- **img**: it is the image in gray scale.

In output there are:

- **mtx**: intrinsic camera matrix.
- **dist**: distortion coefficients.
- **rvecs**: it is tuple that represent the position of the calibration pattern with respect to the camera coordinate space.
- **tvecs**: it is translation vector.

In particular, the algorithm:

1) compute the intrinsic parameters and distortion coefficients.
2) estimate the initial camera pose, with the solvePnp.

3) it runs the global Levenberg-Marquardt optimization algorithm to minimize the re-projection error, it applies the least square between the distance of observed image points and the projected object points.

2.4 Pre-processing of image

Before starting the detection of markers , it is necessary to do a pre-processing of the detected images, in order to remove noise and useless details on the images. This process is equal in all three applications (myMarkers, QrCode,Aruco).
After the detection of video, in all three cases it has been used a D-Link camera, so the video is detected with the function ,provided by OpenCV library:

```
1 cap=cv2.videoCapture(rtsp://admin:dlinkcamera@192.168.0.110:554/live2.sdp)
```

Where this command is used to create an Videocapture object and can read from the camera, in fact the parameters represent the protocol used, the name of camera, the IP address in order to access to camera, and finally the stream. Then with the function *read* is possible to capture frame by frame.

```
1 success, image =cap.read()
```

The first process is to convert the detected image from the original color space RGB (R^3), where R,G and B are matrices, to gray scale (R). Usually is better to use a 32-bit images in applications that need the full range of colors or that convert an image before an operation and then convert back, because with 8-bit images during conversion it is possible lost some information, in fact before the conversion need to scale the image (image = 1/255), in this way there are only values between 0 and 1, and there are not high values.

```
1 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

2.4.1 Smoothing images

The first image processing operation is the *blurring* or *smoothing*. In general in order to do a blurring is necessary to apply on image a filter, usually it is linear:

$$g(i,j) = \sum_{k,l} f(i+k, j+l) \times h(k,l)$$

where the *h(k,l)* is the **kernel** and it represents the coefficients of the filter, and the output *g(i,j)* is the weighted sum of input pixel values.

Gaussian blur

In the application is used a *Gaussian filter*, it is not the fastest filter, but it is the most efficient. It is a *low-pass filter* so attenuates the high frequency signals. The Gaussian smoothing is a 2D - convolution operator, and can be represented as :

$$G_0(x,y) = Ae^{\frac{-(x-\mu_x)^2}{2\sigma_x^2} + \frac{-(y-\mu_y)^2}{2\sigma_y^2}}$$

Where:

- μ: is the mean.
- **A**: it represents the coefficient $\frac{1}{2\pi\sigma^2}$.
- **x**: it is the the distance from the origin in the horizontal axis.
- **y**: it is y is the distance from the origin in the vertical axis
- σ^2: it is the variance for each variables **x** and **y**.

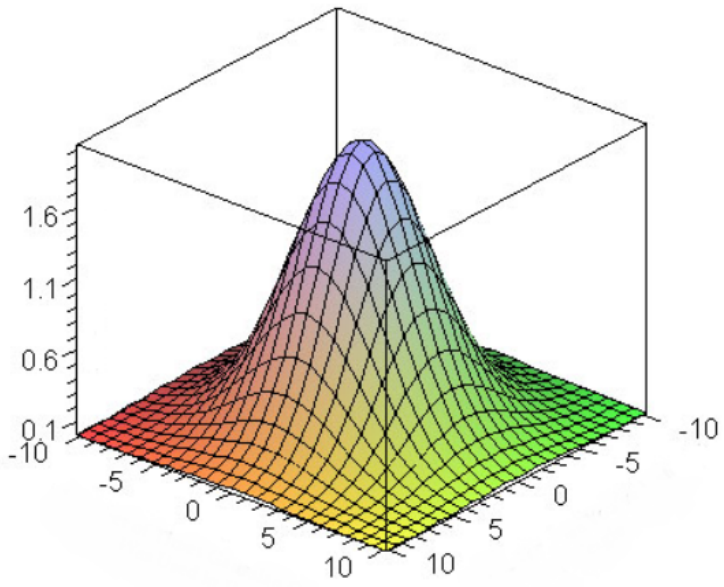


Figure 2.2: 2-dimensional Gaussian function
[9]

In general, the Gaussian filter is similar to mean filter, the only difference is in the **kernel**, in this application it has been chosen a kernel of dimension 5x5 (it is possible see the kernel in Fig. 2.3), and the values of standard deviation for x and y axis are equal to zero. Below, there is the code used to apply the *Gaussian Blurring*:

```
1  blurred = cv2.GaussianBlur(gray, (5, 5), 0)
```

In the Fig. 2.4 is shown how is applied the Gaussian kernel, it is the case with a kernel 3x3, but the process is the same. This is done by placing the center pixel of the kernel on the image pixel and multiplying the values in the original image with the pixels in the kernel that overlap. The values resulting from these multiplications are added up and that result is used for the value at the destination pixel. Looking at the image 2.4, the value at (0,0) in the input matrix is multiplied by the value at (i) in the kernel matrix, and so on, then the results of this multiplications are added up and is obtained as output the pixel (1,1).

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 2.3: Gaussian kernel 5x5
[10]

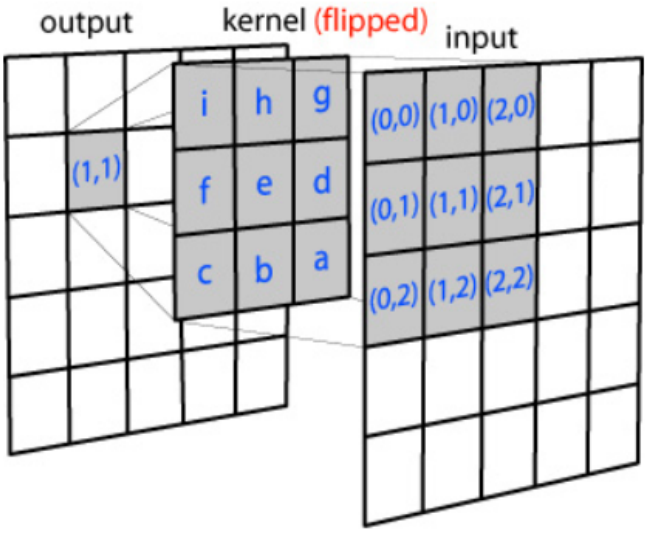


Figure 2.4: Gaussian blurring process
[11]

2.4.2 Edge detection

The edge detection is a set of mathematical models that have as goal to identify points where there are a sharply changes on brightness or there are discontinuities. In myMarkers application it has been used the **Canny edge detection**.

Canny Edge detection

The inventor of this algorithm is John Canny, and this algorithm is divided into different stages [12]:

1) **Reduction of noise**: this phase it has been done applying the Gaussian filter.

2) **Searching the intensity of gradient in the image**: this is possible applying *Sobel Kernel* in both directions (vertical and horizontal).

3) **Non-maximum suppression**: all pixels of the new image is scanned in order to delete all pixels that are not edge.

4) **Hysteresis Thresholding**: this process is done in order to understand which points are edge and which are not.

Searching the intensity of gradient in the image (2), applying the Sobel Kernel 3x3 convoluted with original image, from this is computed the derivatives in both directions, $\mathbf{G_x}$ represents the fist derivative in horizontal direction and $\mathbf{G_y}$ is the first derivative on vertical direction, they represent 2 different images, where along the x coordinate the values increase along the right direction, and the y along down directions. In order to compute the magnitude of gradient and its directions for each pixel:

$$Edge\_Gradient(G) = \sqrt{G_x^2 + G_y^2}$$
$$Angle(\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Non-maximum suppression (3), after the magnitude of gradient and its directions for each pixel is known, the image is scanned pixel by pixel, in order to remove the pixel that are not in edge. In the Fig. 2.5 is shown the process, in particular the **point A** belongs to the edge and it is in the vertical direction of gradient, knowing that the direction of gradient (yellow line) is perpendicular to the edge (red line), so also the **points B** and **C** are analyzed, they are located on the direction of the gradient. After this, the **point A** is compared with the **points B** and **C** to see if it is a local maximum. If it is the local maximum, then it is considered in the next stage, otherwise it is removed, being set to zero. This process is done for all pixels of image.

Hysteresis Thresholding (4), as said before this last stage is useful in order to understand which is a real edge and which not, in order to do this it is necessary to choose two thresholds, maximum value and minimum value, for each pixel is compared its gradient with these two thresholds (Fig. 2.6):

- **Gradient** > **maxValue**: in this case the point is sure a real edge (like the point A).
- **Gradient**< **minValue**: in this case this pixel is discarded because it is not a real edge(like point C).
- **minValue** < **Gradient** < **maxValue**: all pixels that have gradient in this range have to do a further analysis, in fact if this pixel is connected to edge it is considered an edge, although it is discarded.

In the application is used the method *Canny* provided by the Library **OpenCV**, the Canny edge detector is applied on the image where has been applied the Gaussian filter (this image is called *blurred*), the two numbers represent the two thresholds for the hysteresis procedure:

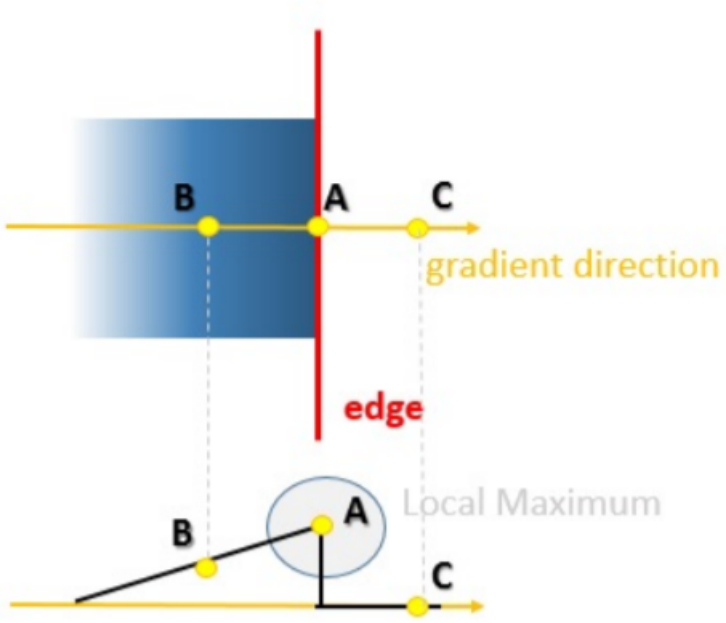


Figure 2.5: Non-maximum suppression
[12]

```
1  im_canny=cv2.Canny(blurred,90,110)
```

In order to improve the quality of image in terms of distinction between black and white colour is used the following method, provided by the **OpenCV** Library:

```
1  threshold=cv2.threshold(gray,100, 255,cv2.THRESH_BINARY)[1]
```

Where the numbers represent the threshold and maximum values, the last parameter represents the type of threshold, it has been chosen the **Binary thresh**:

$$\texttt{dst}(x,y) = \begin{cases} maxValue & \text{if } \texttt{image(x,y)} > \texttt{thresh} \\ 0 & \texttt{otherwise} \end{cases}$$

In the Fig. 2.7 is shown the image processing in application with myMarkers Library, where in the image on the top-left correspond to the image after **conversion in gray scale**, on the top-right is the image after **Canny edge detector**, on the lower part on the left is the image after the **Gaussian blurring** and finally is the image after the **thresholding**.

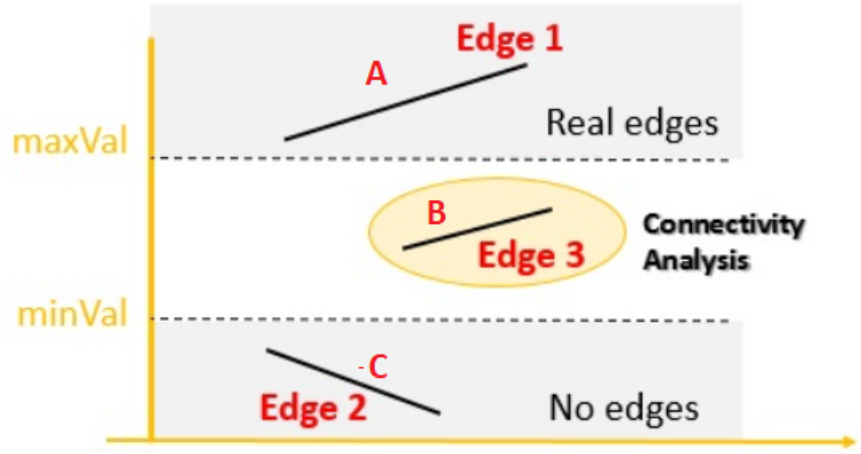


Figure 2.6: Hysteresis Thresholding
[12]

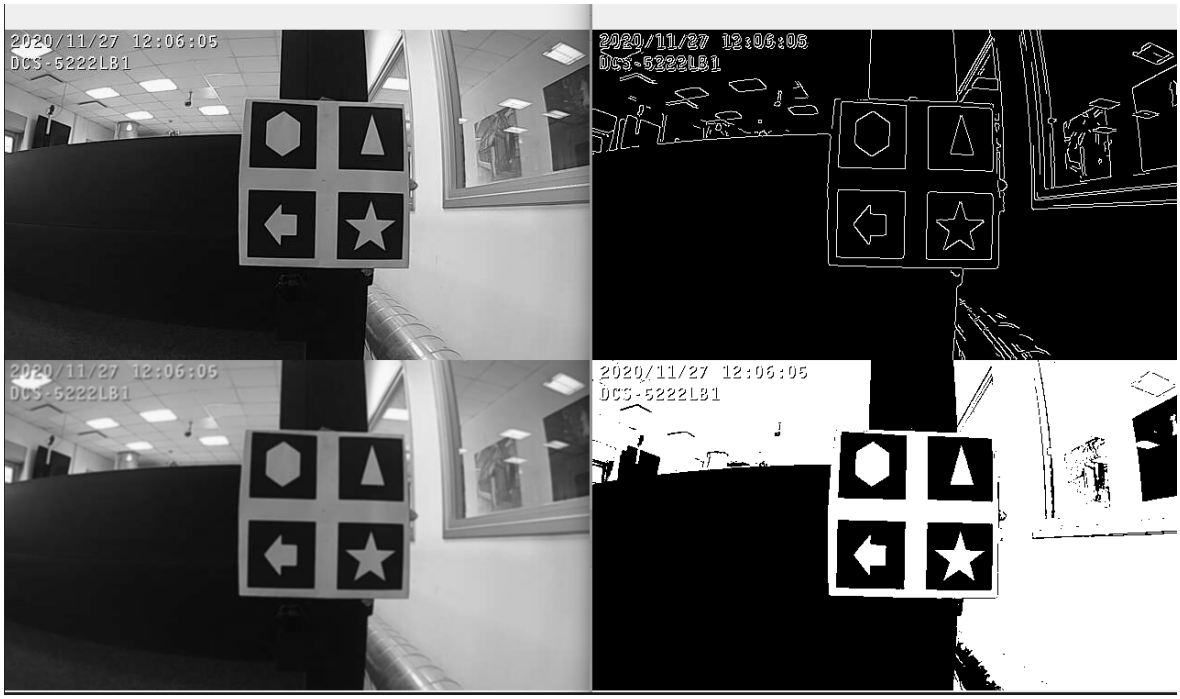


Figure 2.7: Image processing in myMakers

2.5 Detection and Recognition of markers

2.5.1 MyMarkers Library

Firstly, it has been chosen some symbols in order to use them as markers. In particular, it is used the following markers (Fig. 2.8).

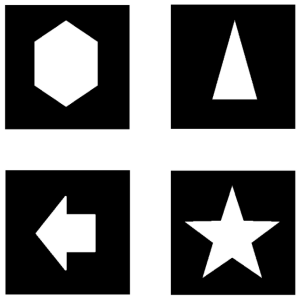


Figure 2.8: myMarkers

In order to detect each markers it has been chosen to used the function that takes the contours and from contours it has been taken only the part of image inside each square, this becomes the region of interest (ROI).

Contours

In order to have better accuracy, it is suggested to use binary images, for this reason before to compute the contours it is applied a *canny edge detection or threshold* (see chapter 2.4).
After it is used the following function, in order to extract contours:

```
1 cnts = cv2.findContours(im_canny,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

where:

1) **im_canny**: it is the image detected by camera after canny edge detection.

2) **cv2.RETR_TREE**: it represents the retrieval mode.

3) **cv2.CHAIN_APPROX_SIMPLE**: it represents contour approximation method.

Retrieval mode

In this case there are some nested figures, this means that there are contours one inside others, so there is a hierarchy. The outer is parent and inner is child. This allows to specify how one contour is connected to each other. How it is possible see in the Fig. 2.9 there are A_0 and A_1 denotes the external and internal contours of the biggest square. So the contour A_0 is parent of contour A_1, it be in hierarchy-0. Similarly contour B_0 is child of contour A_1 and it comes in next hierarchy. Finally contour B_1 is the child of contour B_0, and they come in the last hierarchy level.

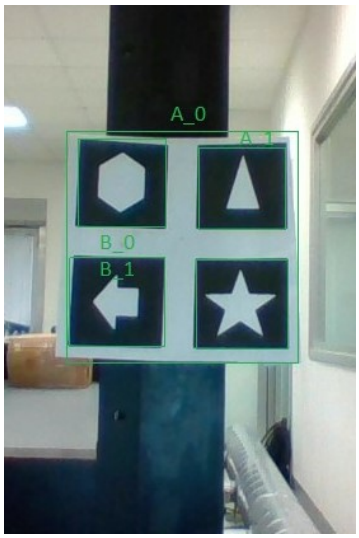


Figure 2.9: contoursHierarchy

Retrivial tree

The method *retrivial tree* returns an array where for each contour is expressed who is the following contour in same hierarchy level, the previous contour, the child and the parent.

Contour approximation

It has been used the method *cv2.CHAIN_APPROX_SIMPLE*, this allows to encode a rectangular contour in 4 points.

2.5.2 Detection

In order to speed up the detection, it has been done a priori filtering in order to exclude all objects, that are not the markers, but are inside the scene. This filtering has been done on the area of contours and has been excluded all elements that are not square. After detecting squares that contain the markers, it has been implemented a function that analyze the detected squares.

2.5.3 Match Patterns

The idea is to use the function **cv2.matchTemplate()**, it slides image of pattern, on the source image, in this case the source image is the image detected by camera. The objective is to find the areas of source image that are very similar to the areas of patch image (the image of pattern). Sliding the patch image on source image means that moving the patch one pixel at a time (left to right, up to down) and for each pixel is computed how much is "good" or "bad" the match at that location respect the source image. For each location is saved on a matrix $\mathbf{R}(\mathbf{x}, \mathbf{y})$ the metric that represents how much "good" is the matching.

```
def matchPattern(self,markersDetected,img_list,image):
    list_result = []
    list_pattern = []
    list_crop = []
    for pattern in img_list:
      pattern_gray = cv2.cvtColor(pattern, cv2.COLOR_BGR2GRAY)
      pattern_res = cv2.resize(pattern_gray, (90, 90))
      iteration=0;
      ratio = image.shape[0] / float(pattern_res.shape[0])
      (h, w) = pattern_res.shape[:2]
      center = (w / 2, h / 2)
      # Perform the rotation
      rot = pattern_res
      for angle in np.arange(0, 360, 45):
        iteration+=1
        M = cv2.getRotationMatrix2D(center, 45, 1.0)
        rot = cv2.warpAffine(rot, M, (w, h))

        blurred = cv2.GaussianBlur(markersDetected, (5, 5), 0)
        threshold = cv2.threshold(blurred, 100, 255,cv2.THRESH_BINARY)[1]
        rot=cv2.resize(rot, (90, 90))

        result = cv2.matchTemplate(threshold,rot, cv2.TM_CCORR_NORMED)

        list_result.append(result)

        list_pattern.append(rot)
        list_crop.append(markersDetected)

    return list_result,list_pattern,list_crop,ratio
```

In particular, there are different type of comparison methods, in this project has been used the **cv2.TM_CCORR_NORMED**. This method used the following type of comparison, given the template image **T** and the source image **I**:

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

where:

- **T(x',y')**: represents the pixel of template image.
- **I(x,y)**: the pixel in the source image.

In order to improve the probability to have higher value of matching, it has been implemented a way to have more comparison with different pose of the pattern images. In fact, each pattern does a rotation of 45° until 360°. The results of matching are saved in the variable **list_result**. The objective is to find the highest value of matching, this

has been implemented extracting the index that contains the highest value in **list_result**. After this, it is necessary to identify which markers has detected, in order to do this it has been implemented the function **detectionPattern**. Before calling the function **detectionPattern**, it is extracted the index of best match:

```
#take the best match
    ind=list_result.index((max(list_result)))]
```

In the variable **list_result** is present the value of each matching, the highest value represents the best match, so with this section of code, shown below, is possible to extract the index where is the best matching.
Knowing that the patterns that need to recognize are 4, and each figure is rotated 8 times, so the variable **list_result** is composed of 32 elements. Each interval represents a symbol:

- **index from 0 to 7**: correspond to the star.
- **index from 8 to 15**: correspond to the triangle.
- **index from 16 to 23**: correspond to the hexagon.
- **index from 24 to 31**: correspond to the arrow.

```
def detectionPattern(self,index,list_cropped_image,peri,approx,
                     contours):
position_crop = 0
list_contours=[]
    for cop in list_cropped_image:
        if (position_crop == index):
            if (index >= 0 and index <= 7):
              if (peri>peri_prev_star and points_star==[]):
                    peri_prev_star = peri
                    points_star=approx
                    shape="star"
                    list_contours.append(contours)
                    break
            if (index >= 8 and index <= 15):
             if (peri>peri_prev_tr and points_tri==[]):
                    peri_prev_tr = peri
                    list_contours.append(contours)
                    shape = "triangle"
                    points_tri=approx
                    break
            if (index >= 16 and index <= 23):
                if (peri>peri_prev_hex and points_hex==[]):
                    peri_prev_hex = peri
                    shape="hexagon"
                    points_hex=approx
                    list_contours.append(contours)
                    break
```

```
            if (index >= 24 and index <= 31):
                if (peri>peri_prev_ar and points_arr==[]):
                    peri_prev_ar = peri
                    list_contours.append(contours)
                    shape="arrow"
                    points_arr=approx
                    break

        position_crop += 1

    return list_contours,points_arr,points_hex,points_tri,points_star
```

2.5.4 QrCode

In the application QrCode is used the **pyzbar** Library, that provide some methods used in order to detect the QrCode used as markers. *Decode* is the method used for the detection of markers, for each QrCode this function returns an object that is composed by different parameters like *data, polygon, type*. The *data* is the parameters that identify the QrCode, each QrCode represents a word or a number, in the following code is represented the technique used to identify some of them, is checked only the first element of vector *data*, because are unique for each QrCode.
The second parameter is *polygon* it is a matrix that represents the 2D coordinates for each corner around the detected QrCode, so there are 4 points for each marker , these 4 points are putted in a single variable called *contours* in order to compute the centre of this square.

```
result = decode(gray)
if result!=[]:
    for i in result:

        if (i.data[0] == 99):  # 3 id
            a = (i.polygon[0][0], i.polygon[0][1])
            b = (i.polygon[1][0], i.polygon[1][1])
            c = (i.polygon[2][0], i.polygon[2][1])
            d = (i.polygon[3][0], i.polygon[3][1])
            contours = [a, b, c, d]
            con = cv2.UMat(np.array(contours, dtype=np.float32))
            M = cv2.moments(con)
            cX = int((M["m10"] / M["m00"]))
            cY = int((M["m01"] / M["m00"]))
            central_3 = (cX,cY)
            cv2.putText(image, "3", (central_3),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 1)


        if (i.data[0] == 112):  # 6 id
          a=(i.polygon[0][0],i.polygon[0][1] )
          b=(i.polygon[1][0],i.polygon[1][1] )
```

```
22            c=(i.polygon[2][0],i.polygon[2][1] )
23            d=(i.polygon[3][0],i.polygon[3][1] )
24            contours = [a, b, c, d]
25            con = cv2.UMat(np.array(contours, dtype=np.float32))
26            M = cv2.moments(con)
27            cX = int((M["m10"] / M["m00"]))
28            cY = int((M["m01"] / M["m00"]))
29            central_6 = (cX,cY)
30            cv2.putText(image, "6", (central_6), cv2.FONT_HERSHEY_SIMPLEX,
                          0.5, (0, 255, 0), 1)
```

In the Fig. 2.10 is possible to see the QrCode used as markers.

Figure 2.10: QrCode markers

2.5.5 ArucoMarkers

Using the Aruco Library is provided methods that allow to detect markers, the method is *detectMarkers*. Knowing the dictionary, it represents the set of markers that will be searched (in this application it is used DICT5X5_1000), the parameter of *adaptiveThreshConstant* is used to set the constant value for adaptive thresholding before finding contours.

```
1  corners, ids, rejectedImgPoints = aruco.detectMarkers(gray,
                        aruco_dict,parameters)
```

The markers detection is performed in the input image and only markers included in the specific dictionary are searched. For each detected marker, it returns the 2D position of its corner in the image in pixel and its corresponding identifier. In the Fig. 2.11 is possible to see the Aruco markers used in this application.

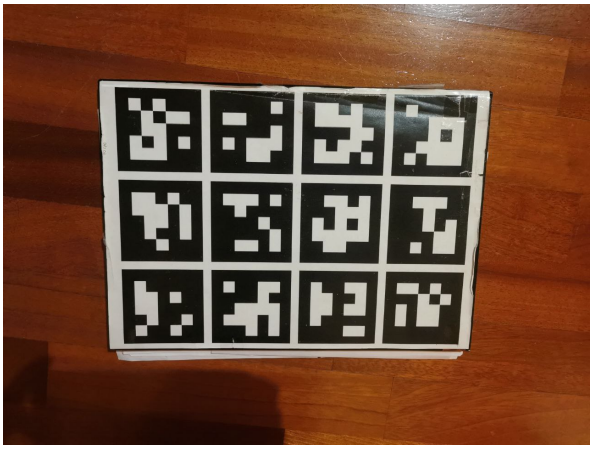


Figure 2.11: Aruco markers

2.6 Position and orientation of markers respect camera

2.6.1 Object Points

First of all, it is necessary to define the objective points, because the computation of the position and orientation depend on them. The object points are 3D points in the coordinate space. The choice of these points is totally free. In these cases they have been chosen with (X,Y,Z) coordinates and Z=0, in this way all points are in the same plane.

2.6.2 Image Points

The image points are 2D points in the image plane. They derived from the detected contours with the function **findContours** and for each detected marker is computed the centroid, with the following code:

```
1  def computeCenter(self, contours):
2      M = cv2.moments(contours)
3      cX = int((M["m10"] / M["m00"]) )
4      cY = int((M["m01"] / M["m00"]) )
5      return cX, cY
```

The image moments is a particular case of image processing in order to find some properties of the image like centroid or area, it is a particular weighted average of the image intensities of pixels.

SolvePnp

The position and orientation of the markers are computed using the function **solvePnp** provided by **OpenCV.**
It is really important specify that for each object point in 3D plane have to correspond an image points in 2D plane. The **solvePnp** works with 4 points or more, but if there are less of 4 points is not possible estimate the position and the orientation.
This function is able to compute the translation and rotation vectors in the camera coordinate frame.
In Fig. 2.12 it is possible see how is estimated the pose of markers from object coordinate frame to camera coordinate frame. The points in world frame $\mathbf{X_w}$ are project into image plane $[\mathbf{u}, \mathbf{v}]$.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{A} \quad \Pi \, {}^{c}\mathbf{T_w} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

where:

1) **A**: it is the the camera intrinsic matrix, that it has been computed during the calibration of camera (see 2.3).

2) $\mathbf{\Pi}$: it represents perspective projection model.

3) ${}^{\mathbf{c}}\mathbf{T}_{\mathbf{w}}$: it is extrinsic parameters, that transform from world to camera frame.

It is important underline that all numbers of these matrices **A** and $\mathbf{\Pi}$ are in functions of camera. At the end, it is possible compute the **rvec** and **tvec** vectors that allow transforming a 3D point expressed in the world frame into the camera frame $\mathbf{X_c}, \mathbf{Y_c}, \mathbf{Z_c}$:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = {}^{c}\mathbf{T}_{\mathbf{w}} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Roll, Pitch, Yaw

The function **solvePnp** returns the **rvec**, it is the rotation vectors, so in order to compute the rotation matrix it is necessary to do a transformation.
From rotation vector to rotation matrix it is necessary apply the **Rodrigues rotation formula**. Given a r vector 3x1 is possible to compute θ, that is the angle of rotation.[14][13]

$$\theta \leftarrow norm(r)$$
$$\mathbf{u} \leftarrow \mathbf{r}/\theta$$
$$R = \cos(\theta)I + (1 - \cos\theta)uu^T + \sin(\theta) \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix}$$

When the rotation matrix is computed it is possible compute the **Roll,Pitch,Yaw angles** :

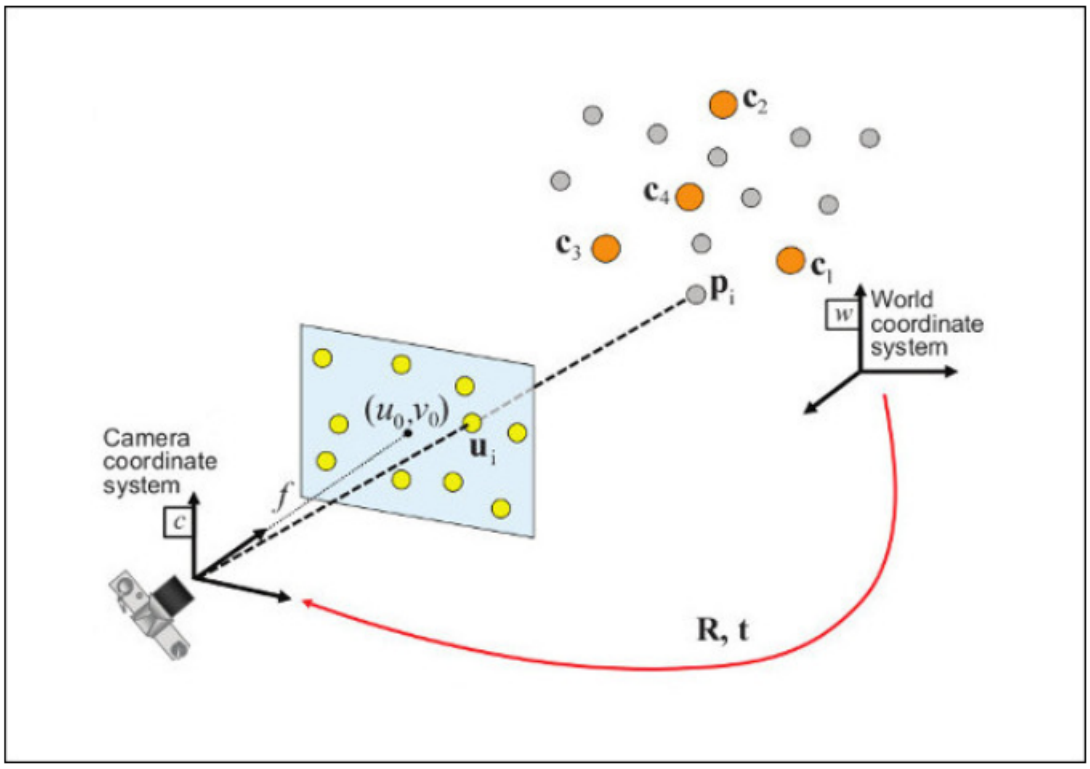


Figure 2.12: solvePnp
[13]

```
1  def rot2Eul(self,R):
2   #theta=beta
3   theta1=math.atan2(-R[2][0],+math.sqrt(R[2][1]**2+R[2][2]**2))
4   theta2=math.atan2(-R[2][0],-math.sqrt(R[2][1]**2+R[2][2]**2))
5   #psi=alpha
6   if math.cos(theta1)!=0:
7      psi1 = math.atan2(R[2][1]/math.cos(theta1), R[2][
                      2]/math.cos(theta1))
8   else:
9      psi2 = math.atan2(R[2][1] / math.cos(theta2), R[2][2] /
                      math.cos(theta2))
10  #phi=gamma
11  if math.cos(theta1) != 0:
12      phi1 = math.atan2(R[1][0] / math.cos(theta1), R[0][0] /
                      math.cos(theta1))
13  else:
14      phi2 = math.atan2(R[1][0] / math.cos(theta2), R[0][ 0] /
                      math.cos(theta2))
15  if phi1==[] and psi1==[]:
16   return[psi2*180/math.pi,theta2*180/math.pi,phi2*180/math.pi]
17  else:
18     return[psi1*180/math.pi,theta1*180/math.pi,phi1*180/math.pi]
```

Where the ψ, θ, ϕ angles correspond to the rotation around x,y,z axis respectively. In particular, if $\theta \in (-\pi/2, +\pi/2)$:

$$\phi = atan2(r_{21}, r_{11})$$

$$\theta = atan2(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2})$$

$$\Psi = atan2(r_{32}, r_{33})$$

It is possible to see a difference between index in the code and in the formula, this difference is due to the fact that in python the index of vectors/matrices starts from 0.

2.6.3 MyMarkers Library

The object points are defined with the following sequence, as shown in the Fig. 2.13.

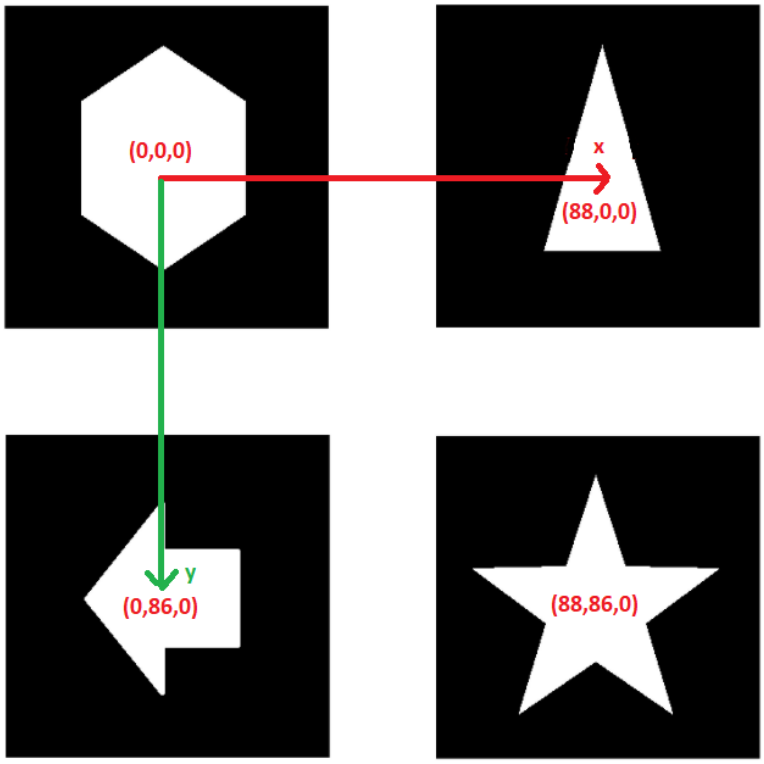


Figure 2.13: Object Points in myMarkers Library

In particular the (0,0,0) coordinates correspond to the centroid of hexagon (number 1), then the centroid of triangle(number 2), the arrow (number 3) and finally the star (number 4). In particular the red line correspond to the direction of x axis, and the green line correspond to the y axis, and the distance between the centroid number 1 and the others are expressed in mm, in order to have more precision. In the following section is possible to see how these object points are declared in Python, they are expressed in *mm*:

```
1 objPoints =[[0, 0, 0], [88,0, 0],[0,86,0],[88,86,0]]
```

2.6.4 QrCode

Using the QrCode library, the object points are defined in different way respect the previous case, in this case there are 6 markers, and the choice of the (0,0,0) point correspond with the centroid of the central marker.
As shown in the Fig 2.14, the red line correspond to the x axis, and the green to the y axis. The distance between the points are expressed in *mm*,as the previous case.

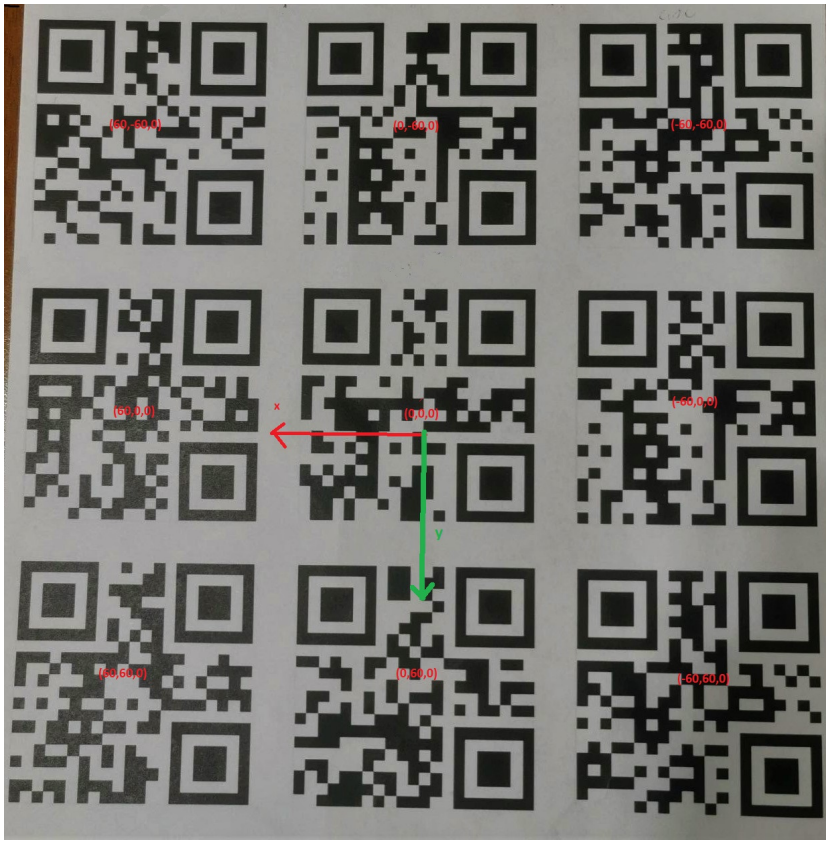


Figure 2.14: Object Points in qrCode Library

Each markers's centroid has an object coordinate, as shown in the Fig. 2.14 from top-left corner to the bottom-right corner:

```
objPoint=[[60,-60,0],[0,-60,0],[-60,-60,0],[60,0,0],
            [0,0,0],[-60,0,0],[60,60,0],
            [0,60,0],[-60,60,0]]
```

2.6.5 Aruco

In Aruco application, the object points are not defined manually, but given a board configuration and a set of detected markers is possible compute the object points. In fact, after the markers are detected, it is checked that these markers belong to the board, and from the board are extracted the information like the image points (2D) and the object points (3D). All of this are implemented in the function **aruco.getBoardObjectAndImagePoints**, it is provided by the Aruco library.

```
1  void getBoardObjectAndImagePoints(const Ptr<Board> &board,
                        InputArrayOfArrays detectedCorners,
2      InputArray detectedIds, OutputArray objPoints, OutputArray imgPoints)
```

It is important notice how the Aruco board is created, it is necessary to know the following parameters:

- **markersX**: number of markers in X direction.
- **markersY**:number of markers in Y directions.
- **markerLength**: it is marker side length (expressed in mm).
- **markerSeparation**: it is the separation between two markers (expressed in mm).
- **dictionary**: it is dictionary of markers indicating the type of markers,in this case it is DICT_5X5_1000.

Chapter 3

Achieved results

In the following section is shown the results achieved for the three different applications. When the application starts to run it is given, to the user, the possibility to choose the point that have to be reached, this point have to express in *cm*, and the orientation that the user wants give to the markers. It is important specify that this point is chosen by the user respect the distance from the camera and the orientation is expressed respect the reference system defined by the declaration of the object points.
After this, there is the calibration of camera and then starts the detection of video.

3.1 Simulation with myMarkers Library

In this simulation is given as target position (0,0,35) cm and the orientation of (0,0,90) degree. Before describing the position and orientation of markers, it is possible to see that the application is able to detect and recognize the markers, in fact the detected markers are surround with a red square, and inside there is written the shape of the detected markers (hexagon,triangle,arrow,star). In the Fig. 3.1 is shown the starting position of markers in this simulation, it is possible to see the reference system that is centred in the hexagon's centroid, where the green line corresponds to the y axis, the red line to the x axis, and the blue ones to the z axis (the orientation is outgoing from camera). The second reference system present in the Fig. 3.1 corresponds to the target orientation that the markers have to reach, in order to do this need to do a rotation of -90° around z axis. In the Fig. 3.1 are shown also the values of the target position and orientation, the actual position x=[-0.667] y=[-6.187] z=[35.8] with the distance from the target x=[0.667] y=[6.187] z=[0.807], for the actual orientation R=[1.51°] Y=[5.986°] Z=[0.910°], with a difference from the target orientation R=[1.51°] P=[5.986°] Y=[-89.08°].
In fact, the actual position and the actual orientation are computed respect the point (0,0,0) of markers that correspond to the centroid of hexagon. At the beginning, this point is higher respect the target point so the coordinate y is negative, also the coordinate x is on the right so it is negative, for the z coordinate it corresponds to

the distance of markers from the camera and it is 35.8 cm. About the orientation the *actual orientation* it correspond to the reference system that it has been declared with the definition of object points, in fact the values of R=[1.51°] Y=[5.986°] Z=[0.910°] are around 0, and in order to reach the target orientation it is necessary to obtain the following values R=[1.51°] P=[5.986°] Y=[-89.08°] (in the Fig. 3.1 correspond to the section *Difference from the target(degree)*.

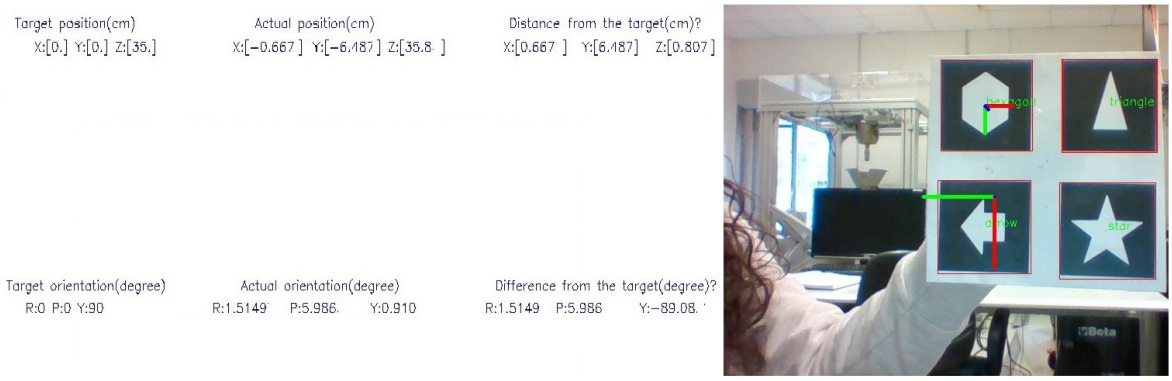


Figure 3.1: Starting point

In the Fig. 3.2 it is possible see how change the reference system of markers after a rotation of 90° degree around z axis.The actual orientation becomes R=[1.97°] P=[7.3987°] Y=[93.123°],so about the orientation it is reached the target, in fact the difference from the target is R=[1.97°] P=[7.3987°] Y=[3.123°], that is really near to 0°, it is important specify that these tests are done manually so it is not easily reachable exactly a difference of 0°. Consequently, after the rotation also the position changes, in fact the origin of reference system becomes x=[8.94] y=[-5.97] z=[38.01] (actual position expressed in *cm*).

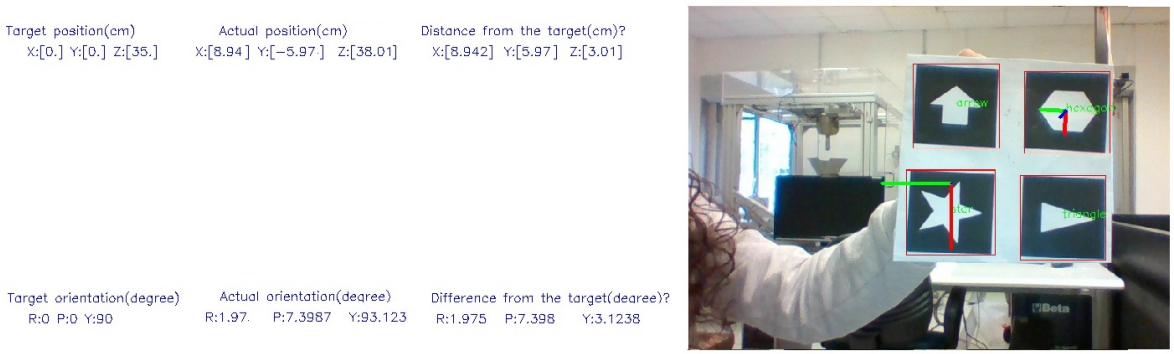


Figure 3.2: After rotation

Now, in order to reach the target it is necessary to do translations until the exact position is reached. As shown in Fig. 3.3, the two reference systems are almost coincident, there is a minimum difference in the axis y and z, in fact the actual position is x=[0.002] y=[-0.406] z=[39.71] (expressed in *cm*), so in order to reach the target it is necessary to bring the markers closer to the camera. About the orientation it has been reached, in fact the difference from the target are R=[0.48°] P=[9.249°] Y=[0.499°].

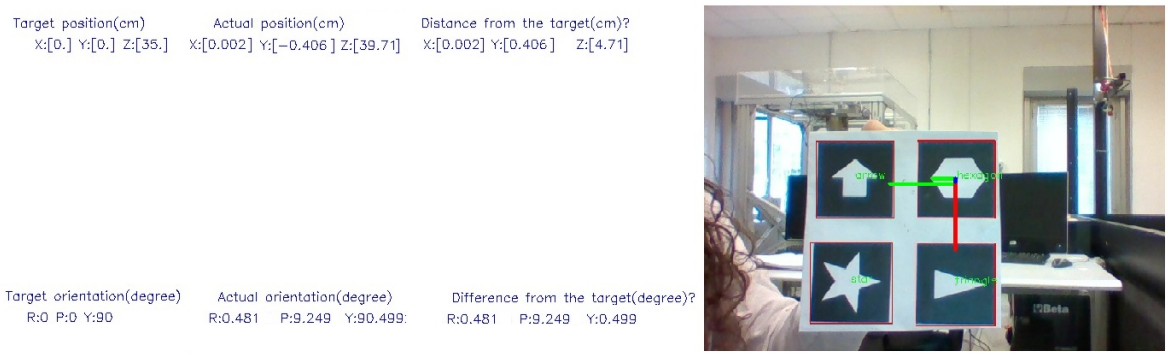


Figure 3.3: Near the target point

Finally,in the Fig. 3.4 is possible see when the target position is reached. The markers have an actual position of x=[-0.129] y=[0.017] z=[35.98] with a distance from the target x=[0.129] y=[0.017] z=[0.98] (all distances are expressed in *cm*). For the the actual orientation is R=[1.32°] P=[8.25°] Y=[90.48°] with a difference from the target R=[1.32°] P=[8.25°] Y=[0.48°].

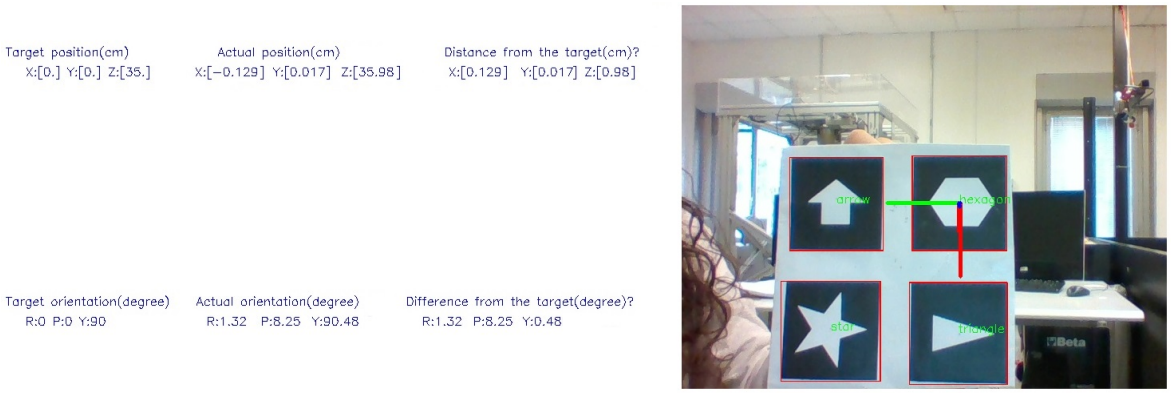


Figure 3.4: Final position

3.2 Simulation with QrCode

In this section it is shown the simulation of application with the QrCode as markers. Firstly, it is important see that the markers are detected and recognize,in fact every time the markers is recognize, above it, is written the number that represents the sequence of markers.

The user has chosen a target position of x=[0] y=[0] z=[30] and target orientation R=[0°] P=[0°] Y=[0°], it is important remember that the positions are expressed in *cm* and the orientation in *degree*. Then, as shown in Fig. 3.5 that it is the first position at the beginning of simulation, where the actual position is x=[-4.83] y=[-0.22] z=[32.256] (expressed in *cm*) with a distance from the target is x=[4.83] y=[0.22] z=[2.256] (expressed in *cm*), in fact the origin of reference system, that has as origin the centroid of the central marker, is on the right respect the target reference system so the x coor-

dinate is negative, about the actual orientation is R=[1.25°] P=[-0.77°] Y=[-0.39°], in fact there is a little bit of difference in the angle around the x axis.

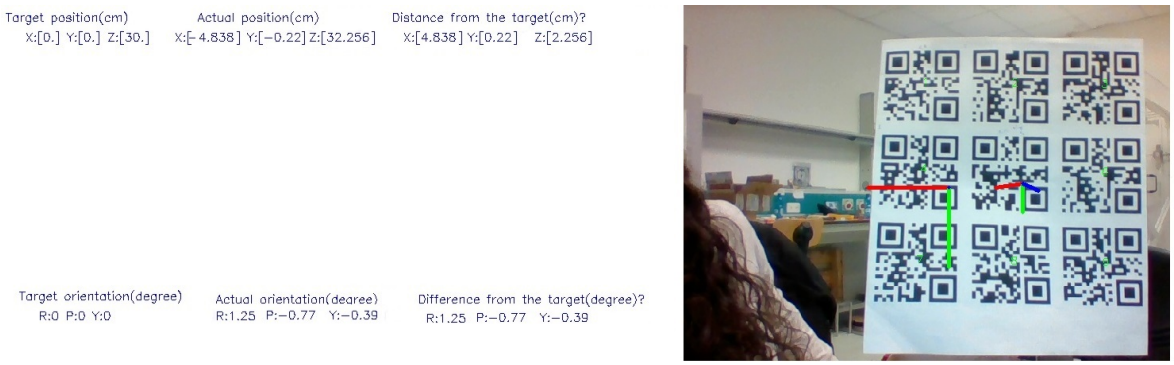


Figure 3.5: Initial position

In the Fig. 3.6 the markers are really near to the target position and the actual position is x=[-0.002] y=[-0.616] z=[29.05] (expressed in *cm*) with a distance from the target x=[0.002] y=[0.616] z=[-0.947] (expressed in *cm*), so in order to reach the target the markers have to go a little bit lower. As is possible see in the figure there is a little bit of rotation around the x,y,z axis with the actual orientation R=[3.42°] P=[-5.327°] Y=[-3.93°].

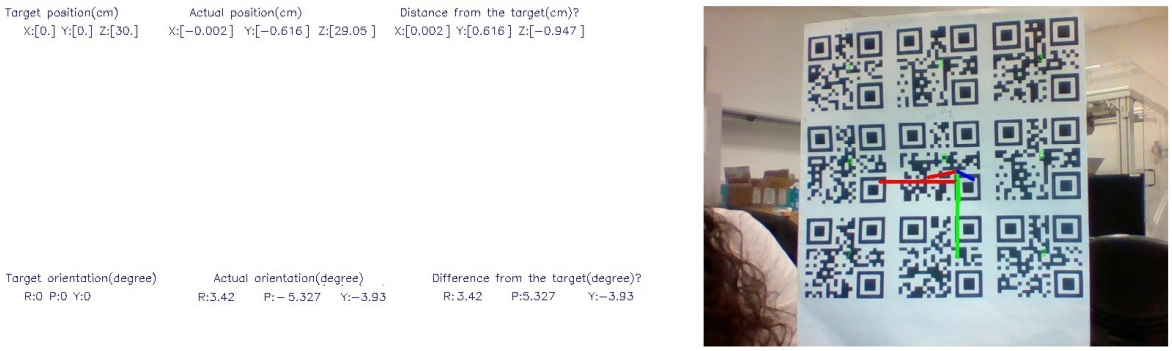


Figure 3.6: Intermediate position

Finally, in the Fig. 3.7 is shown when the target is reached. The actual position is x=[0.053] y=[0.04] z=[30.81] (expressed in *cm*), so the distance from the target is x=[0.053] y=[0.04] z=[0.81] (expressed in *cm*), and an orientation of R=[1.30°] P=[11.22°] Y=[-0.99°],in fact in the photo is possible to see that the axis x and z are not really coincident with the correspondents axis of target reference system in fact there is an angle of 11.22° around axis y.

3.3 Simulation with Aruco

In this section is possible to see a simulation of Aruco application. With Aruco markers the reference system of markers is positioned in the upper-right angle of the marker with first ID. The target position is x=[0] y=[0] z=[60] with an orientation R=[0°] P=[0°]

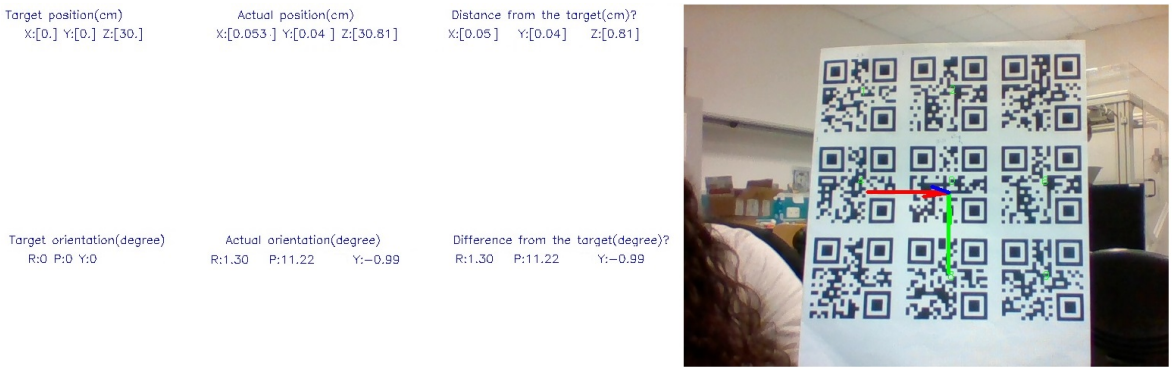


Figure 3.7: Final position

Y=[0°], all measures for the distance are expressed in *cm* and for the orientation are in degree.
In the Fig. 3.8 is shown the initial position of markers where the distance from the actual position and the target position is x=[13.746] y=[16.97] z=[7.59] (expressed in *cm*), this means that in order to reach the target it needs to increase the distance from the camera (z axis) and moves in the correct direction for the x and y axis.About the orientation is possible to see that respect the target orientation it is necessary to do a rotation of 180° around z axis(see in the Fig. 3.8 the column *Difference from the target(degree))*.

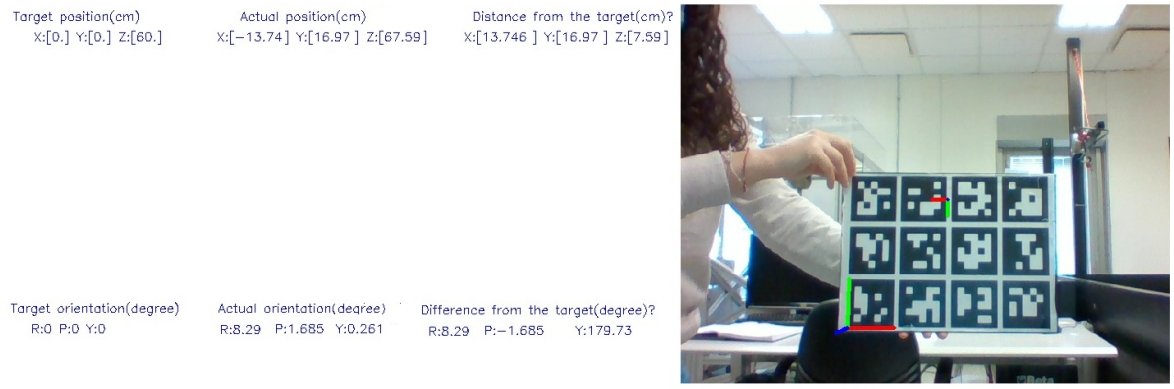


Figure 3.8: Initial position

In the Fig. 3.9 shown the values of position and orientation after a rotation of 90°around z axis. The actual position is x=[10.12] y=[-10.86] z=[61.61](expressed in *cm*). About the orientation there is the difference for the angle around the z axis, in fact they are R=[10.41°] P=[-5.74°] Y=[-91.02°], so need the last rotation around z axis.
The Fig. 3.10 shows the last rotation ,in fact now the orientation is equal to the target orientation there is only small difference due to the fact that all tests have been done manually, the difference from the target is R=[0.268°] P=[-3.76°] Y=[1.09°]. Now need to reach the target position in terms of distance, because how is possible to see in the Fig. 3.10 the distance from the target is x=[12.39] y=[2.37] z=[-0.10] (expressed in *cm*).

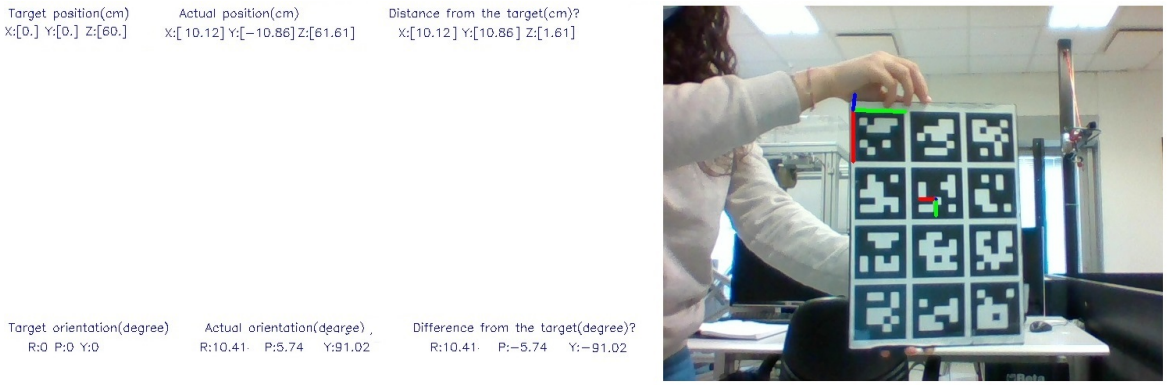


Figure 3.9: After 90° of rotation around z

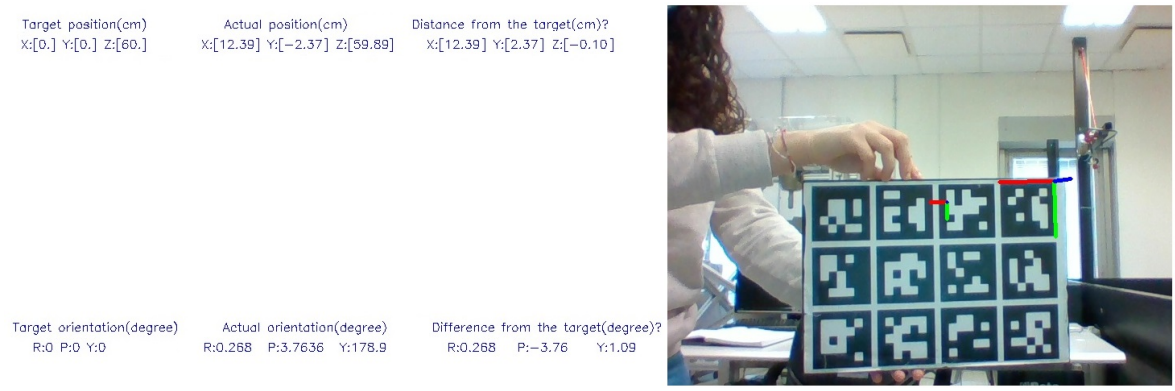


Figure 3.10: After second rotation of 90°around z

The Fig. 3.11 shows the moment when is reached the target point with an actual position x=[0.26] y=[0.63] z=[60.64] (expressed in *cm*). About orientation there is only a small difference for the angle around y axis, in fact the difference from the target is R=[0.60°] P=[-5.45°] Y=[0.58°].

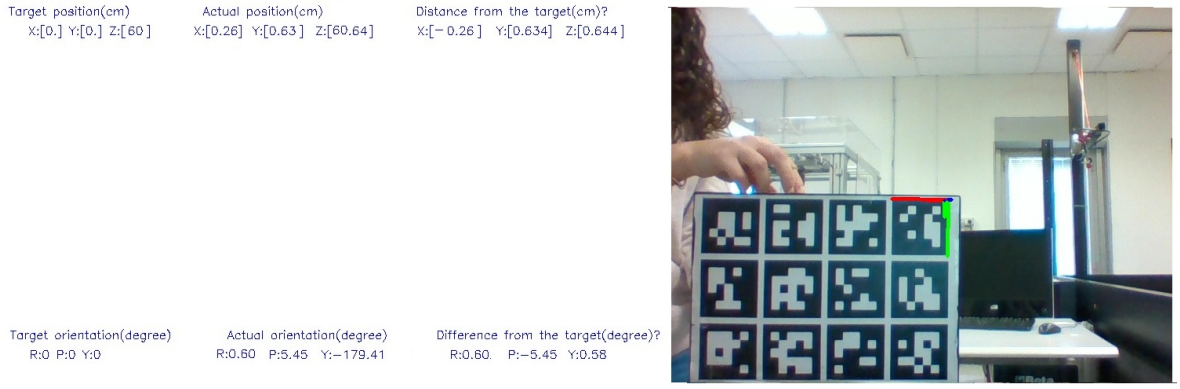
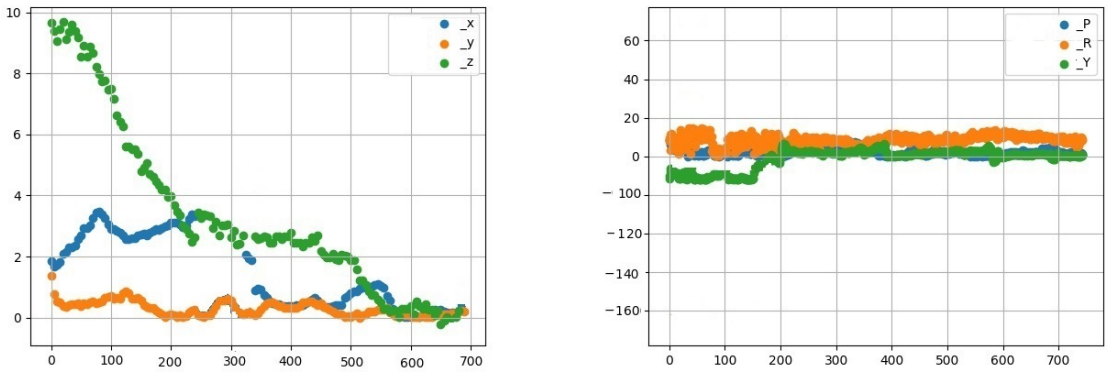


Figure 3.11: Final position



(a) Variations of translation with myMarkers

(b) Variations of Orientation with myMarkers

Figure 3.12: Graphics that show variations during simulation with myMarkers Library

3.4 Comparisons between the different applications

3.4.1 myMarkers

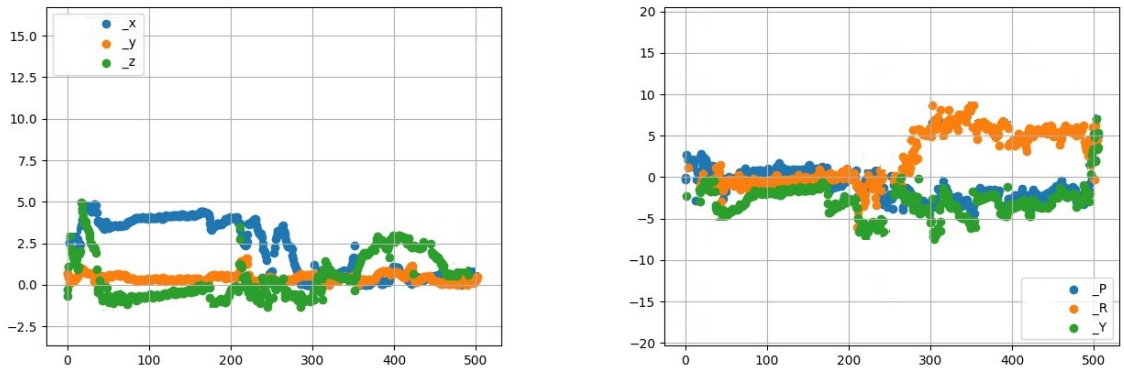
In the Fig. 3.12 is possible see the variations of distance from the target position [Fig 3.12a](expressed in cm) and the difference from the target orientation [Fig 3.12b] (expressed in degree) during the simulation, along x axis is represented the duration of simulation. It is possible see that at the beginning there is a little bit of difference on the z axis, but then following the correct movements to reach the target position all goes to 0, the same for the orientation where the difference is -90° at the beginning, then after the rotation around z axis becomes 0.

3.4.2 QrCode

In the Fig. 3.13 is possible to see the variations of distance during simulations with QrCode (Fig 3.13a) and the variations of orientations of markers during simulation (Fig 3.13b), along x axis is represented the duration of simulation. In the Fig. 3.13a is shown the translation, where is possible to see that at the end the difference is around zero. In the Fig. 3.13b shows how change the orientation and it is possible see that for Pitch angle there is a difference of 11.22° and only at the and there is some sample that shows that a difference of 0° is reached.

3.4.3 Aruco

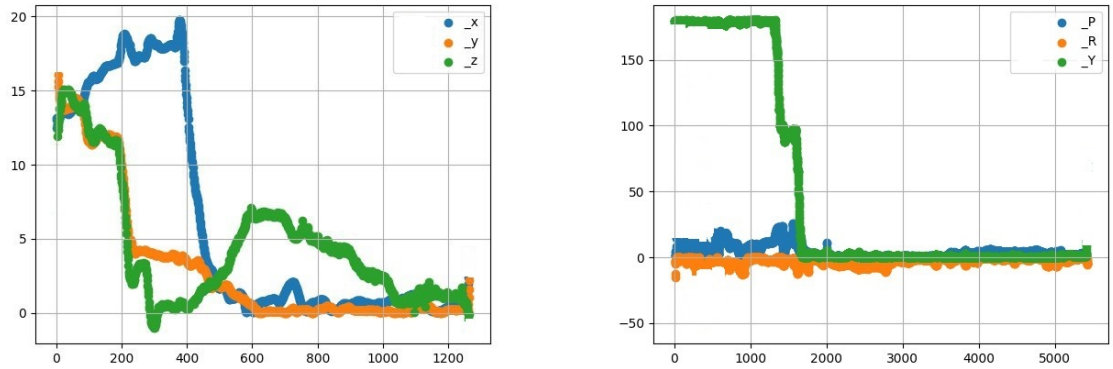
Also in this Fig. 3.14 is possible to see how change, during the simulation, the distance from the target (3.14a) and the difference of orientation from the desired orientation (3.14b). In 3.14b is possible see that at the beginning there is a difference of 180° around z axis, and at the end after two rotations of 90° around z axis the difference is



(a) Variations of translation with QrCode

(b) Variations of Orientation with QrCode

Figure 3.13: Graphics that show variations during simulation with qrCode



(a) Variations of translation with Aruco

(b) Variations of Orientation with Aruco

Figure 3.14: Graphics that show variations during simulation with Aruco

around zero.

3.4.4 Pro and Cons

It is possible to do a comparison in terms of recognition of markers, estimation of position and orientation respect the camera.
The application with myMarkers Library is not the best choice for the recognition of markers, this is due to the symbols used for the markers (Fig. 2.8), because the ideal markers have to be asymmetrical, in fact with the symbols used in this case becomes very difficult the recognition, and it works correctly in ideal conditions of light and with small distance from the camera.
This is not a problem for Aruco and QrCode, in these cases the recognition works also if the distance from camera is high.
If the recognition of markers works correctly also the estimation of position and orientation is good. In fact for the application with myMarkers Library is not able to

estimate the position and orientation if one of the markers is not detected and recognized. For the other two applications the estimation of positions and orientations works correctly and have more robustness, because there are more markers. Even if the application with QrCode is not able to follow a faster rotation of markers, instead the Aruco is able to do this.

Library	Recognition	Position	Orientation
myMarkers	small distance and in good condition in terms of light	good	good with slow rotation
QrCode	good(also high distance)	good	good with slow rotation
Aruco	good (also high distance)	good	good with fast rotation

Table 3.1: Summary Pro e Cons in the three applications

3.5 Improvements in Aruco

In order to improve the application where has been used Aruco, it has been added a further priori check. Knowing that the detection and recognition works better when there are optimal conditions in terms of light and when markers are not really far from the camera, in order to increase robustness is checked if the camera detects 2 or more equal markers, this is interpreted as an error, the estimation of position and orientation is not computed, the application starts to compute the position and orientation when all detectedmarkers are different among them, so when the video return in optimal conditions. In the Fig. 3.15 is shown in the case when it is detected 2 equal markers, they are surrounded with a red square.

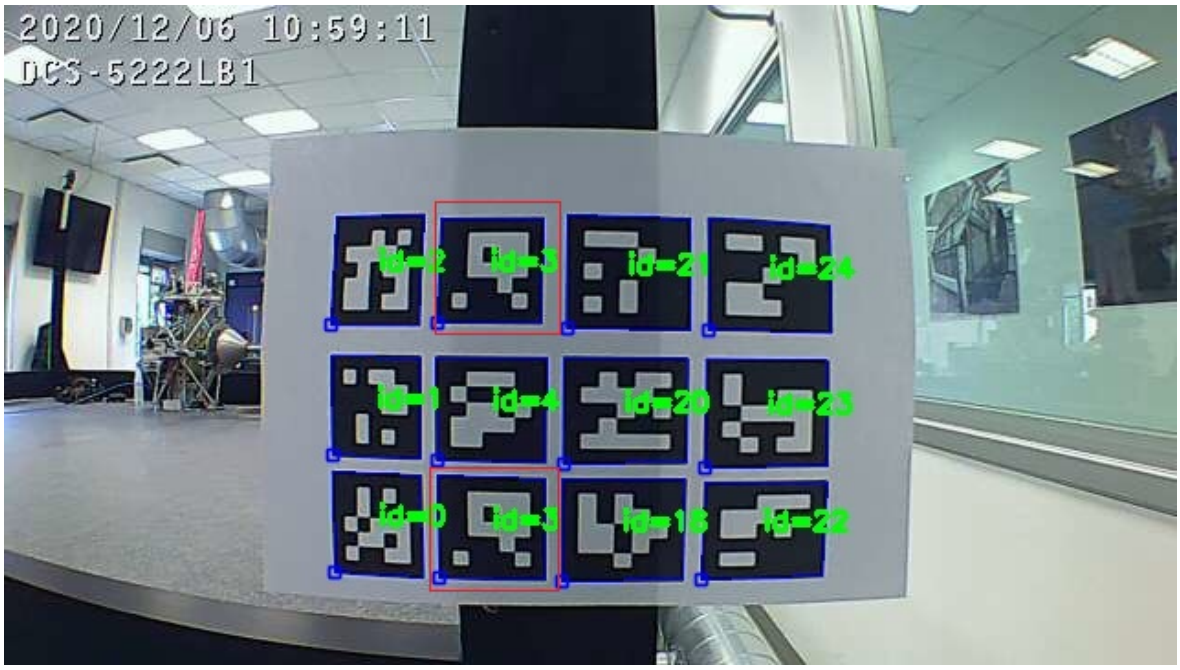


Figure 3.15: Duplicated markers

Part II

Second Part

Chapter 4

Model based

In the last years, the problem of object recognition and its 6D pose estimation has received considerable attention. Nowadays, there are a lot of applications that needs an efficient recognition and pose estimation approaches, like autonomous driving, robotics vision. These kinds of approaches are based on shape, geometry of the object for this reason are called **model based** or **shape based**. The difference with the methods seen in the first part is that with this approach is not necessary used markers around the object, so it removes this limitation.

4.1 State of art

In order to do the **3D object recognition** , is very common do the *template matching*. This method consists into compare the run time image with database images of the object, which has to be recognized. However, this method has some limitations, it is not efficient for a real-time applications, because requires a great computational effort, and this method is very efficient for heterogeneous objects, but for industrial parts, which are usually symmetric and have simple, homogeneous shapes, it is not effective. Another method for the object recognition is the region based approaches usually it is used for object recognition and pose estimation on 2D images. It consists to extract the contours of the object, in order to exploit the *contours features*.[15]
About the **3D pose estimation** there are different approaches, one of them is the method proposed by Brachmann et al.[16] they estimate the 6d pose of an object from a single RGB image. The idea is to reduce the uncertainty in object coordinates and object prediction iteratively, in order to deal with missing depth information. Another approach is to estimate 3D pose and segmentation of rigid 3D objects using a single monocular RGB camera based on local color histograms.[17] It works very well in cases of cluttered backgrounds, heterogeneous objects and occlusions.
In the last years, with the advent of artificial intelligence, scientists have been integrate computer vision with Deep Learning and Artificial Neural Networks, in order to improve the accuracy. Like [18] used both colour images and depth features to train a

Convolutional Neural Network (CNN) model. These methods based on Neural Network requires a richer training set, it is an essential part for learning the algorithm. The method [19] which is a real-time 2D object detector. This network works on 3D point clouds. However, point clouds are usually noisy and capturing a clean one requires more expensive computation power. Other approach, in order to estimate 3D pose, used some algorithm of point set registration,like ICP,CPD. In conclusions, applying the Neural Network or Deep Learning, in order to estimate the 3D pose, improves the accuracy of estimation but it introduces some limitations , one of these is the quantity of data used to create a properly data-sets can require a lot of consuming time, another reason is that these methods are not a robust technique for industrial parts where different parts can be very similar in shapes.[15]
Taking account the finality of the work, it has been chosen to not use Neural Network or Deep Learning, because for space applications they require a lot of computing time, so it has been done an application that extracts sets of 3D point cloud, only in a specific region of the object, in this way is deleted noise and useless part, and has been implemented three different algorithm used for the point set registration, in order to see the comparison among them. In the following paragraph will be presented ,in detail, the implemented method.

4.2 Implemented method

In this section is described the approach used to estimate the 3D pose of known object. One application has been done with the Intel Realsense T265 (4.1), it is a tracking camera using a stereo vision with two fisheye lenses. Second application has been

Figure 4.1: Tracking camera T265 (credits: Intel)

done using the Depth camera D435 (4.2), it has a stereo vision and RGB camera. The approach used in order to estimate the homogeneous transformation matrix is the

Figure 4.2: Depth camera D435 (credits: Intel)

same, the only difference is on the extractions of the 3D points.

It has been taken an image of the object, it has been considered as reference, and the run-time images, detected by camera. The run time image has been compared with this reference image and in both images are extracted the features inside the region of interest (ROI), in this case the ROI is the regions inside the circle (4.3). After the

Figure 4.3: Features extracted inside ROI

extractions of features, it has been done a match between the key-points on the run-time and on reference image, only after matching it has been passed from 2D points to 3D points, the technique used for the transformation from 2D to 3D points is different in the two cameras, more details about this is given in the sections (4.5) and (4.6).
After the computation of these sets of matched 3D points, it has been computed the transformation matrix from reference image to run time. In order to do this, it has been implemented three different algorithms:

1) **Homography estimation**: it computes the Homography matrix with 3D points.

2) **ICP (Iterative Closest point)**: it is a method based on distance.

3) **CPD (Coherent Point Drift)**: it is probabilistic method for registration of point sets.

The last two algorithms are methods used in **pairwise Point Set Registration**, the goal is to find the best transformation matrix that best aligns the two sets of 3D points.

4.3 Object detection

The object used in this work is showed in figure (4.4), it represents in small scale a LAR (launch adapter ring) of a satellite, with two bolts. In order to recognize it in the scene is used an approach similar to the method used in the first part. In particular, it

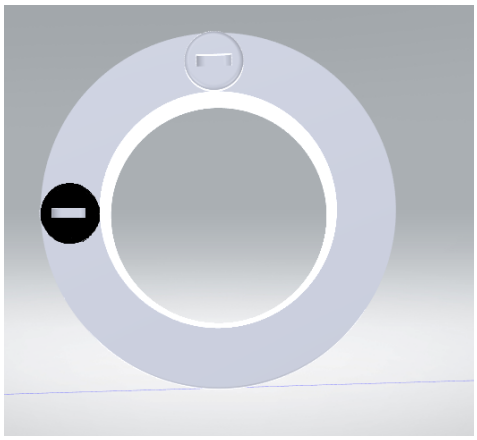


Figure 4.4: Object to recognize(CAD)

Figure 4.5: On the left the image with Gaussian filter, and on the right the image after Canny edge detection

has been done all pre-processing image described on 2.4 (Gaussian blur, Canny Edge Detection) , see figures 4.5.

After the reduction of noise, it has been applied the function, provided by OpenCV, **findContours** (more details in subsection *Contours* on chapter 2.4). Then is applied the function **approxPolyDP**, this function approximate the input curve or polygon to an approximated curve with less points, it uses Ramer–Douglas–Peucker algorithm. Computed this approximated curve , it is converted from **Point2f** to **vpImagePoint** , using the function **convertFromOpenCV**, provided by **VISP**. These vector of *vpImagePoint* is used in order to declare the region of interest (**ROI**). In fact this ROI needs in order to do the tracking of the object and the features are extracted only inside this region.

4.4 Tracking of object with Visp

After the first detection of object , the modular library**Visp** [20] allows to do a good tracking of object, it is able to follow the object even with fast movements. This is

done using the module **Moving edge tracker**, it is able to track an object exploiting its contours, like line or ellipse. In this work it has been used the classes **vpMe** and **vpMeEllipse**. With the following code, it has been initialize the moving edges parameters used later by the ellipse tracker. The tracking is done along the normal of the contour with a range of 25 pixels. For each pixel along the normal we will compute the oriented convolution. The pixel that will be selected by the moving edges algorithm will be the one that has a convolution higher than 15000. Between two consecutive moving edges on the contour we keep a space of 2 pixels:

```
1 vpMe me;
2 me.setRange(25);
3 me.setThreshold(15000);
4 me.setSampleStep(2);
```

Then is initialize the tracker, in this case **vpMeEllipse** that will track the ROI of the object. The tracker is initialized with the previous moving-egdes parameters.(line 2). Without any no user intervention has been initialized the tracker through the vpImagePoint computed in the previous section:

```
1 vpMeEllipse ellipse;
2 ellipse.setMe(&me);
3 ellipse.initTracking(image,curves);
```

Then in order to do a run time tracking, every time the camera detects new image, the tracker is update using the function **ellipse.track(image)**, this function recomputes all parameters of curve , and search the new points of contours on the new image thanks to the Moving edge parameters. The figure 4.6 shows the tracking run time of the object:

Figure 4.6: Tracking with Visp, the red points are the features extracted

4.5 Project using camera T265

The camera T265 is a stereo camera with 2 fisheye lens.

4.5.1 Calibration

The method used to calibrate camera is really similar to the method described in the section 2.3, in fact firstly it has been computed the object points and the image points of chessboard on first and second lens. Using the function, provided by OpenCV,*fisheye::stereoCalibrate*, it has been computed the camera intrinsic matrix, the distortion vector composed by four elements, the rotation matrix and the translation vector in order to pass from first lens to the second. In the following code is possible see that there are also another parameter , it is the flag that has been used for the calibration, it set the skew coefficient (alpha) to zero.
In order to find the undistorted points, it has been necessary rectify the fisheye cameras. This it has been done using a function provided by OpenCV *fisheye::stereoRectify*, it receives as input the intrinsic matrices for left and right lens, their distortion coefficients, the rotation matrix and the translation vector from first to second lens, and computed as output the 3x3 rectification transform for first and second lens, the 3x4 projection matrix in the new (rectified) coordinate systems for the first and second camera. The flag is set, this means that the function makes the principal points of each camera have the same pixel coordinates in the rectified views, the last two parameters represents the new focal length and the divisor for the new focal length.

```
1 //Calibration of stereo fisheye
2 cv::fisheye::stereoCalibrate(object_points,left_img_points,right_img_point,
3 K1,D1, K2, D2, Size(848, 800), R,T, CALIB_FIX_SKEW,
4 TermCriteria(TermCriteria::COUNT + TermCriteria::EPS, 100, 1e-5));
5 //Rtification of stereo fisheye
6 cv::fisheye::stereoRectify(K1, D1,K2,D2, Size(848, 800), R, T, R1,
7                            R2, P1, P2, CALIB_ZERO_DISPARITY,
8                            Size(848, 800), 0.0, 1.1);
```

In the figure 4.7 is shown the difference between an distorted image (on the left) and undistorted image after calibration (on the right).

4.5.2 Extractions 3D points

This camera is a tracking camera, so in order to obtain the 3D points it is necessary do triangulation exploiting the stereo vision. Firstly has been taken a photo of object, with left and right lens, these 2 photos has been taken as reference. After the preprocessing of these images, it has been extracted features using **ORB** as *detector* and as *extractor* . Then it has been done a matching, using as *matcher* the **Brute Force**, between the features of left and right image. The parameter of **Brute Force matcher** is the norm, it represents the distance measured between descriptors, in this case that

Figure 4.7: Difference between original image taken by fisheye(left) and the same image undistorted after calibration(right)

the extractor is an **ORB** the norm types is **Hamming Norm**. The matching is done using a *knnMatch*, it founds the best two matches.
Then run time has been taken the images from left and right lens, the process is the same of process that it has been described for the reference images. After this, it has been done the lastly matching between the descriptors on the left run time image and left reference image, this last matching is fundamental in order to establish which key-points on run time image corresponds with the key points on reference image, in reality it is possible do the matching between right run time image and right reference image, because the descriptors, that will match with the descriptors of reference image, of left run time image and right run time image are the same, because they are the descriptors after matching left-right run time image. In the following figure 4.8 is described the process of matching. At the end of matching process, it has been obtained four sets

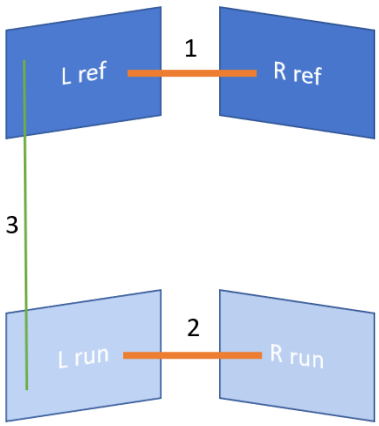


Figure 4.8: Summary of matching process

of 2D points ,two sets are the 2D points on image plane of left and right reference images, the other two sets are the 2D points for the left and right run time images. It

is important specify that the extractions of features is done only one time for the left and right reference images, the only extraction that is done iteratively is on the left and right run time images.
It has been necessary to transform the 2D points to 3D points, in order to do this it has been done **triangulation**. This process it has been done using the functions provided by OpenCV. However, before the triangulation it is important to delete the distortion, because both lens are fisheye, so firstly the 2D points on image plane are projected on the undistorted plane and after is applied the triangulation. In order to applied these two functions **undistortPoints** and **triangulatePoints** it has been necessary used the parameters founded with calibration.

```
1 //Delete distortion on 2D points on left image
2 cv::fisheye::undistortPoints(2d_distorted_left,2d_undistorted_left,K1,
                        D1, R1, P1);
3 //Delete distortion on 2D points on right image
4 cv::fisheye::undistortPoints(2d_distorted_right,2d_undistorted_right,K2,
                        D2, R2, P2);
5
6 //Triangulation obtaining 3D points
7 cv::triangulatePoints(P1, P2, 2d_undistorted_lef,2d_undistorted_right,
                        pnts3D);
```

4.6 Project using camera D435

4.6.1 Calibration

The depth camera D435 is a stereo camera, but it has not a fisheye lens, so the calibration is really simple, it has been used the auto calibration provided by the library of Intel Realsense. The parameters founded after the calibration are the focal length f_x, f_y and the optical centers c_x, c_y for the RGB camera and Depth camera, the distortion coefficients are zeros.

4.6.2 Extractions 3D points

The process of extractions are really similar to the process seen for the camera T265, however using this camera the transformation from 2D to 3D points is easier, because this camera is a depth camera so the triangulation it has been done automatically.
Firstly, it has been taken a photo from RGB camera, this image is taken as reference image, and the extractions of features is done from the RGB camera, but the descriptors are taken on gray image. The type of extractor is the same of extractor used in the subsection 4.5.2.
Then, the same it has been done with run time image, always using RGB camera. The matching has been done between the descriptors extracted from reference image taken by RGB camera and the descriptors taken from the run time image always from RGB

camera. In the figure 4.9 is possible see an example of matching.

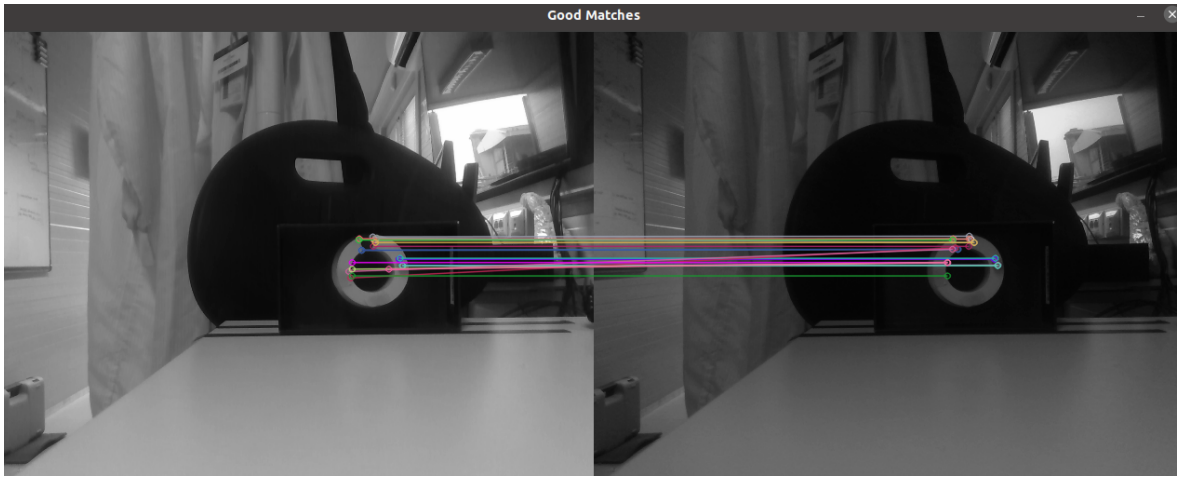


Figure 4.9: Example of matching between run time and reference image

The **transformation from 2D to 3D** has been done align the image taken by the depth camera to the color camera, in this way every pixel of image taken by RGB correspond to the same number of pixel on depth camera, the particularity is that the pixels on RGB image has only two information (x,y) , instead each pixel of image taken by depth camera has three information (x,y,z). In the following code is possible see that after the matching there are two sets of 2D points, each point has two coordinates x and y that correspond a specific key-point on color camera image, so it has been searched the pixel, in the image of depth camera, with the nearest x and y coordinates to the matched key-point taken by RGB camera.

```
1  //Definition of maximum difference between the coordinates x and y of
                      pixel on depth image and pixel on rgb image
2  vector<float> errors_x_run(list_pixel_vp_run.size(), 0.5);
3  vector<float> errors_y_run(list_pixel_vp_run.size(), 0.5);
4  for (unsigned int i = 0; i < I_depth_raw.getHeight(); i++) {//rows
5     for (unsigned int j = 0; j < I_depth_raw.getWidth(); j++) {//columns
6         if (I_depth_raw[i][j]) {
7         //depthScale is known parameter of camera
8             float Z = I_depth_raw[i][j] * depthScale;
9             depthMap_run[i][j] = Z;
10            for (int u = 0; u < list_pixel_vp_run.size(); u++) {
11                if (abs(list_pixel_vp_run.at(u).get_i() - i) <
                        errors_x_run.at(u) and
12                    abs(list_pixel_vp_run.at(u).get_j() - j) <
                        errors_y_run.at(u)) {
13                    errors_x_run.at(u) =
                        abs(list_pixel_vp_run.at(u).get_i() - i);
14                    errors_y_run.at(u) =
                        abs(list_pixel_vp_run.at(u).get_j() - j);
15                    index_run.at(u).x = i;
16                    index_run.at(u).y = j;
17                    index_run.at(u).z = Z;
```

```
                }
            }

        } else {
            depthMap_run[i][j] = 0;
        }
    }
}
```

In the following section is showed how transforms the 2D points to 3D points, in the previous step it has been extracted the depth for each pixel, and now with the function **rs2_deproject_pixel_to_point** provided by the library Intel, given the intrinsic matrix of depth camera and the coordinates x and y of pixel and also its depth, it is given as output the 3D points:

```
///deprojection
for (int u = 0; u < index_ref.size(); u++) {
   float depth_pixel[2];
   float depth_point[3];
   depth_pixel[0] = index_run.at(u).y;
   depth_pixel[1] = index_run.at(u).x;
   rs2_deproject_pixel_to_point(depth_point, &depthIntrinsic,
                    depth_pixel, index_run.at(u).z);

   float depth_pixel_ref[2];
   float depth_point_ref[3];
   depth_pixel_ref[0] = index_ref.at(u).y;
   depth_pixel_ref[1] = index_ref.at(u).x;
   rs2_deproject_pixel_to_point(depth_point_ref, &depthIntrinsic,
                    depth_pixel_ref, index_ref.at(u).z);
   if(depth_point_ref[2]!=0 and depth_point[2]!=0 ){
       points3d_ref.push_back(Point3f(depth_point_ref[0],
                    depth_point_ref[1], depth_point_ref[2]));
       points3d_run.push_back(Point3f(depth_point[0], depth_point[1],
                    depth_point[2]));
   }
}
```

4.7 Computation of transformation matrix

4.7.1 Homography estimation

Given two sets of 3D points $\overrightarrow{x_{ref}} \in \Re^{4,N}$, it represents the set of extracted 3D points on reference image, and $\overrightarrow{x_{run}} \in \Re^{4,N}$, the sets of extracted 3D points on run time image, where N is the number of points presents in both sets. The objective is the computation of matrix $H(x, \theta)$:

$$\overrightarrow{x_{ref}} = H(x, \theta) * \overrightarrow{x_{run}}$$

It is necessary to insert a possible error $\varepsilon \in \Re^{4,N}$:

$$\overrightarrow{x_{ref}} - H^*(x,\theta) * \overrightarrow{x_{run}} = \varepsilon$$

The homography estimation with 3D points has 12 unknown variables, considering only one point for each set:

$$\begin{bmatrix} x_{ref} \\ y_{ref} \\ z_{ref} \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & t_x \\ a_{21} & a_{22} & a_{23} & t_y \\ a_{31} & a_{32} & a_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{run} \\ y_{run} \\ z_{run} \\ 1 \end{bmatrix}$$

The unknown variables are $a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}, t_x, t_y, t_z$. In order to increase robustness, this algorithm has been implemented with *Ransac*, the computation of Homography matrix it has been done with random subset of 3D points consisting of at least 6 points, so the dimension n of these subsets is 6:

$$\begin{bmatrix} x^1_{ref} & y^1_{ref} & z^1_{ref} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & x^1_{ref} & y^1_{ref} & z^1_{ref} & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x^1_{ref} & y^1_{ref} & z^1_{ref} & 0 & 0 & 1 \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ x^6_{ref} & y^6_{ref} & z^6_{ref} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & x^6_{ref} & y^6_{ref} & z^6_{ref} & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x^6_{ref} & y^6_{ref} & z^6_{ref} & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \\ a_{33} \\ t_x \\ t_y \\ t_z \end{bmatrix} = \begin{bmatrix} x^1_{run} \\ y^1_{run} \\ z^1_{run} \\ . \\ . \\ . \\ . \\ x^6_{run} \\ y^6_{run} \\ z^6_{run} \end{bmatrix}$$

Calling the first matrix with the reference points $\mathbf{A} \in \Re^{3n,12}$,the matrix with unknown variables $\mathbf{X} \in \Re^{12,1}$, and the matrix with run time points $\mathbf{b} \in \Re^{3n,1}$, at the end the objective is the computation of $\mathbf{X}$:

$$X = (A^T * A)^{-1} * A^T * b$$

At the end, it is necessary apply **SVD** in order to transform the matrix into orthonormal matrix. Using *Ransac*, the new transformation matrix $\mathbf{H}^*$ is applied to all points(N) of both sets, in order to compute the errorε , and after 100 iterations,it has been chosen the transformation matrix with minimum error. The following code shows how is built the matrix A, with a subset of 3D points *cloud*:

```
1  void computeMatrixA(vector<Point3f> cloud){
2     A= Eigen::MatrixXf::Zero(3*cloud.size(),12);
3     int j=0;
4     int count=0;
5     for(int i=0;i<cloud.size();i++){
6         if(count==0){
```

```
            A(j,0)=cloud.at(i).x;
            A(j,1)=cloud.at(i).y;
            A(j,2)=cloud.at(i).z;
            A(j,9)=1;
        }
        j++;
        count=count+1;
        if(count==1){
            A(j,3)=cloud.at(i).x;
            A(j,4)=cloud.at(i).y;
            A(j,5)=cloud.at(i).z;
            A(j,10)=1;
        }
        j++;
        count=count+1;
        if(count==2){
            A(j,6)=cloud.at(i).x;
            A(j,7)=cloud.at(i).y;
            A(j,8)=cloud.at(i).z;
            A(j,11)=1;
        }
        j++;
        count=0;
    }
}
```

The following code shows how is built the matrix B, with a subset of 3D points *cloud run*:

```
void computeMatrixB(vector<Point3f> cloud_run){
    int j=0;
    B_init= Eigen::MatrixXf::Zero(3*cloud_run.size(),1);
    for(int i=0;i<cloud_run.size();i++){
       B_init(j,0)=cloud_run.at(i).x;
       j++;
       B_init(j,0)=cloud_run.at(i).y;
       j++;
       B_init(j,0)=cloud_run.at(i).z;
       j++;
    }
}
```

Finally this section shows the computation of Homography matrix, if the rank of matrix is not full this means that a singularity is reached, so the algorithm can not work:

```
bool computeHomography(){
   Eigen::FullPivLU<Eigen::MatrixXf> decomp((A.transpose()*A).inverse());
   if(decomp.rank()!=12){
       cout<<"Singularity"<<endl;
       return false;
      }
   else{
```

```
8           Eigen::MatrixXf m=((A.transpose()*A).inverse()*A.transpose())*B;
9           this->H_run<<m(0,0),m(1,0),m(2,0),m(9,0),
10                  m(3,0),m(4,0),m(5,0),m(10,0),
11                  m(6,0),m(7,0),m(8,0),m(11,0),
12                  0,0,0,1;
13          return true;
14      }
15  }
```

4.7.2 Iterative Closest Point

This algorithm has been proposed by Arun et al.[21], it is a pairwise point set registration based on distance method, because minimize the difference between the 2 sets of point cloud.
Given two sets points where $p_i^{'}$ is the set of 3D points on run time image, and p_i 3D points on reference image. The objective is to find $\mathbf{R}\in \Re^{3,3}$ and $\mathbf{t}\in \Re^{3,1}$ that minimize:

$$\sum^2 = \sum_{i=1}^{N} ||p_i^{'} - (R * p_i + t)||^2$$

The algorithm consists to compute:

1) The centroid of all points on reference image ($\mathbf{p}$) centroid and with all points on run time image ($\mathbf{p}^{'}$):

$$p = \frac{1}{N}\sum_{i=1}^{N} p_i \text{ [Reference centroid]}$$

$$p^{'} = \frac{1}{N}\sum_{i=1}^{N} p_i^{'} \text{ [Run time centroid]}$$

2) The distance from its centroid for each point:

$$q_i = p_i - p \quad \forall i = 1, ..., N \text{ [Distance for each point on reference set]}$$

$$q_i^{'} = p_i{'} - p^{'} \quad \forall i = 1, ..., N \text{ [Distance for each point on run time set]}$$

3) The 3X3 matrix $\mathbf{H}$:

$$H = \sum_{i=1}^{N} q_i * (q_i^{'})^T \qquad \forall i = 1, ..., N$$

4) Applying the **SVD** on H matrix:

$$H = U \bigwedge V^T$$

where:

the rotation matrix is $\mathbf{R} = V * U^T$
the translation vector is $\mathbf{t} = p^{'} - R * p$

This algorithm works if the *det(**R**)* is equal to +1 , if it is equal to -1 this means that algorithm fails. In order to reduce the uncertainty , the centroid on reference and run time 3D sets point has been computed not as mean point but as median. The following code shows the computation of median, for each coordinates x,y,z it is computed separately:

```
// Function for calculating median
double computeMedian(vector<float> array){
    size_t size = array.size();
   if (size == 0){
       return 0;  // Undefined, really.
   }
   else{
       sort(array.begin(), array.end());
       if (size % 2 == 0){
           return (array[size / 2 - 1] + array[size / 2]) / 2;
        }else{
              return array[size / 2];
       }
    }
}
```

After the computation of centroid, it is compute the distance from each centroid, the variable k if is equal to 0 it refers to the 3D points on reference image, if is 1 it is for the 3D points on run time image:

```
void computeDistance(vector<Point3f> cloud,Point3f centroid,int k,int N){
   for (int i=0;i< N ;i++){
     if(k==0){
      errReference.push_back(Point3f(cloud.at(i).x-centroid.x,
      cloud.at(i).y-centroid.y,cloud.at(i).z-centroid.z));
    }
    if(k==1) {
       errRunTime.push_back(Point3f(cloud.at(i).x-centroid.x,
       cloud.at(i).y-centroid.y,cloud.at(i).z-centroid.z));
    }
  }
}
```

Then is computed the homogeneous matrix:

```
void computeH(vector<Point3f> errReference,vector<Point3f> errRunTime){
   Eigen::MatrixXf q(3, 1);
   Eigen::MatrixXf q_first(1, 3);

   Eigen::Matrix3f sum;
   sum << 0, 0 ,0,
```

```
            0, 0, 0,
            0,0 , 0;
    Eigen::MatrixXf a(3, 3);
    if(errRunTime.size()==errReference.size()){
        for(int i=0;i<errRunTime.size();i++){
            q<< errReference.at(i).x,
                    errReference.at(i).y,errReference.at(i).z;
            q_first<< errRunTime.at(i).x,errRunTime.at(i).y,
            errRunTime.at(i).z;

            Eigen::MatrixXf a=q*q_first;
            sum=sum+a;
        }
    }
    this->H=sum;
}
```

Finally is applied the **SVD** to H matrix, and is found the *rotation matrix* and *translation vector*:

```
bool computeSVD(){
    Eigen::JacobiSVD<Eigen::MatrixXf> svd(this->H, Eigen::ComputeFullU |
                    Eigen::ComputeFullV);
    Eigen::MatrixXf L=svd.matrixV();
    this->X=L*svd.matrixU(); //da init to run;
    int det=this->X.determinant();
    Eigen::FullPivLU<Eigen::MatrixXf> lu_decomp(this->X);
    int rank=lu_decomp.rank();
    if(det==1 and rank==3){
       Eigen::MatrixXf p(3, 1);
       Eigen::MatrixXf p_first(3, 1);
       p << this->centroidReference_MEDIAN.x,
                    this->centroidReference_MEDIAN.y,
                    this->centroidReference_MEDIAN.z;
       p_first << this->centroidRunTime_MEDIAN.x,
                    this->centroidRunTime_MEDIAN.y,
                    this->centroidRunTime_MEDIAN.z;
       this->tvec = p_first - (this->X * p);
       this->eulerAngle1 = this->X.eulerAngles(0, 1, 2);//x,y,z
       return true;
    }
    else
       return false;
}
```

4.7.3 Coherent Point Drift

CPD is probabilistic method,where the objective is to recover the transformation that maps one point set to the other, it works for rigid and non-rigid point set registration. The alignment of two point sets is considered as a probability density estimation

problem. The GMM centroids is forced to move coherently as a group to preserve the topological structure of the point sets [22].
The parameters used are:

- **D**: it is the dimension of the points sets, in this case the points sets are 3D, so D=3.
- **N,M**: are the number of points in the point sets (reference and run time), in this case the two sets have the same number of points, so N=M.
- $\mathbf{X_{NxD}}$: it is the set of 3D points extracted by reference image.
- $\mathbf{Y_{MxD}}$: it is the set of 3D points extracted by run time image.
- $T(Y, \theta)$: it is the transformation matrix 4x4.

The run time points set **Y** is considered the GMM centroids, and the reference points set **X** is the data points generated by the GMM. The algorithm follows these steps:

- **Initialization**: at the first iteration the rotation matrix **R** is initialized to identity matrix and translation vector **t** to zero. There are some parameters that have been chosen a priori, one of this is the scale factor **s** , in this case it is equal to 1 because the object has always the same dimension, the second factor is the parameter **w**, it represents the amount of noise in the point sets, it is the ratio of outliers. In particular, a wrong value of this parameter can cause a wrong result of algorithm. It has been chosen a value of 0.35, it means that there are on average 5 outliers on a total of 14 points. The third parameter is the minimum value of threshold , if σ^2 reached this value, the convergence it has been reached, this means that if the value of minimum threshold is smaller the accuracy is higher but it requires more time.
  It is computed the σ^2:

$$\sigma^2 = \frac{1}{D*N*M} \sum_{n=1}^{N} \sum_{m=1}^{M} \|\mathbf{x_n} - \mathbf{y_m}\|^2$$

- **Expectation-maximization algorithm**: **EM** is an iterative method, it is divided into two step, **E-step** (*expectation*) guess the values of parameters and using *Bayes'* theorem to compute *a posteriori* probability distributions of mixture components,computing **P** matrix $\in \Re^{M,N}$:

$$p_{mn} = \frac{e^{-\frac{1}{2\sigma^2}\|\mathbf{x_n} - (s\mathbf{R}\mathbf{y_m} + \mathbf{t})\|^2}}{\sum_{k=1}^{M} e^{-\frac{1}{2\sigma^2}\|\mathbf{x_n} - (s\mathbf{R}y_k + \mathbf{t})\|^2} + (2\pi\sigma^2)(\frac{D}{2})\frac{w}{1-w}\frac{M}{N}}$$

  the **M-step** (*maximization*) ,that minimizing the expectation of negative log-likelihood function [23], founds the *"new"* values of these parameters:

  - $\mathbf{N_p} = \mathbf{1^T P 1}$, where $N_p \in \Re^{1,1}$, $\mathbf{1} \in \Re^{N,1}$ is a column vector of ones, consequently $1^T \in \Re^{1,M}$.

- $\mu_x = \frac{1}{N_p}\mathbf{X^T P^T 1}$, where $\mu_x \in \Re^{3,1}$, $\mathbf{X^T} \in \Re^{3,N}$ is the transpose matrix where on rows there are the coordinates *x,y,z* and on columns all points extracted on reference image.
- $\mu_y = \frac{1}{N_p}\mathbf{Y^T P 1}$, where $\mu_y \in \Re^{3,1}$, $\mathbf{Y^T} \in \Re^{3,M}$ is the transpose matrix where on rows there are the coordinates *x,y,z* and on columns all points extracted on run time image.
- $\hat{\mathbf{X}} = \mathbf{X} - \mathbf{1}\mu_x^T$, where 1 is the column vector of ones.
- $\hat{\mathbf{Y}} = \mathbf{Y} - \mathbf{1}\mu_y^T$, where 1 is the column vector of ones.
- $\mathbf{A} = \hat{\mathbf{X}}^\mathbf{T}\mathbf{P}^\mathbf{T}\hat{\mathbf{Y}}$, computing **SVD** on $\mathbf{A} = \mathbf{USSV^T}$, the rotation matrix is: $\mathbf{R} = \mathbf{UCV^T}$, where **C** is a diagonal matrix with values of 1,1,...., $det(\mathbf{UV^T})$
- The translation vector $\mathbf{t} \in \Re^{3,1} : \mathbf{t} = \mu_x - s\mathbf{R}\mu_y$
- Finally is update the value of $\sigma^2 : \sigma^2 = \frac{1}{N_p * D}(tr(\hat{\mathbf{X}}^\mathbf{T} d(\mathbf{P^T 1})\hat{\mathbf{X}}) - s \quad tr(\mathbf{A^T R})$, where $d(\mathbf{P^T 1})$ is a diagonal matrix.

The EM algorithm proceeds until the convergence is reached, this means that the σ^2 has reached the minimum value chosen a priori in the initialization. In order to increase robustness this algorithm has been wrapped with *Ransac*, where **CPD** is applied for 70 iterations on a randomic subset of at least 8 points on reference and run time point clouds, and then in order to compute the best transformation matrix is computed the *error of projection*:

```
1  //compute error on all points cloud and cloud_run
2   float sum_err = 0;
3   vector<Point3f> list_error;
4   for (int i = 0; i < points3d_ref.size(); i++) {
5     Eigen::MatrixXf init(3, 1);
6     init << points3d_run.at(i).x,
7             points3d_run.at(i).y,
8             points3d_run.at(i).z;
9     Eigen::Vector3f run_test_single = computation.R_best* init +
                      computation.t_best;
10    float err_x = points3d_ref.at(i).x - run_test_single[0] ;//x
11    float err_y = points3d_ref.at(i).y- run_test_single[1];//y
12    float err_z =  points3d_ref.at(i).z- run_test_single[2] ;//z
13
14    list_error.push_back(Point3f(err_x,err_y,err_z));
15    //computation of norm
16    float norm = (err_x * err_x) + (err_y * err_y) + (err_z * err_z);
17    sum_err = sum_err + norm;
18  }
```

At the end only the **Rotation matrix** and **translation vector** with the minimum projection error is taken.
The following code shows the initialization of σ:

```
1  //dimension D=dimension of point sets(3), N number of points on source
                      data set, M number of points in target data set
2  void computeSigma(int N,int M,vector<Point3f> src,vector<Point3f> dst){
3     float sum=0;
4     for(int i=0;i<src.size();i++){
5         for (int j=0;j<dst.size();j++){
6             //computation norm
7             float x=src.at(i).x-dst.at(j).x;
8             float y=src.at(i).y-dst.at(j).y;
9             float z=src.at(i).z-dst.at(j).z;
10            sum=sum+(x*x)+(y*y)+(z*z);
11        }
12    }
13    sigma_square=sqrt(sum)/(N*M*3);
14 }
```

This code shows the implementation of **EM algorithm**:

```
1  void CPD(vector<Point3f> src,vector<Point3f> dst,int D,int M,int
                      N,Eigen::MatrixXf matrix_src,Eigen::MatrixXf
                      matrix_dst){
2     this->local_P.resize(M,N);
3     this->local_P.setZero();
4     int counter=0;
5     do{ //until we reach the convergence
6     //////E-step
7         Eigen::Vector3f x_n;//source
8         for(int i=0;i<src.size();i++){
9             float denominator=0.0;
10            float inter= (this->w ) / (1.0 - this->w);
11            denominator= powf(2 *3.14 * sigma_square, 1.5) *inter;
12            for(int k=0;k<dst.size();k++) {
13
14                Eigen::Vector3f y_k;//destination
15                x_n << src.at(i).x,
16                       src.at(i).y,
17                       src.at(i).z;
18
19                y_k << dst.at(k).x,
20                       dst.at(k).y,
21                       dst.at(k).z;
22                Eigen::Vector3f vect=(this->R * y_k + this->t);
23                float norms=powf(x_n[0]-vect[0],2.0)+
                       powf(x_n[1]-vect[1],2.0)+ powf(x_n[2]-vect[2],2.0);
24                denominator = denominator+expf(-1.0/(2*sigma_square)*
                       sqrt(norms)) ;
25            }
26            for (int m  = 0; m <dst.size(); m++) {
27                Eigen::MatrixXf y_m(3,1);//destination
28                y_m<<dst.at(m).x,
29                     dst.at(m).y,
```

```
30                      dst.at(m).z;
31                x_n<<src.at(i).x,
32                      src.at(i).y,
33                      src.at(i).z;
34                Eigen::Vector3f sol;
35                sol= this->R*y_m+this->t;
36                float norms=powf(x_n[0]-sol[0],2.0)+
                        powf(x_n[1]-sol[1],2.0)+ powf(x_n[2]-sol[2],2.0);
37                float p_mn = expf((- 1/(2.0*this->sigma_square) )*
                        sqrt(norms));
38                this->local_P(m, i) = (p_mn / denominator);
39             }
40     }
41     ////////////// M-Step
42     Eigen::MatrixXf row_one;
43     row_one.resize(1,N);
44     Eigen::MatrixXf column_one;
45     column_one.resize(M,1);
46     for (int i=0;i<M;i++) {
47         row_one(0, i) = 1.0;
48         column_one(i,0)=1.0;
49     }
50
51     // N_p dimension 1x1
52     Eigen::MatrixXf Np_matrix=row_one*this->local_P*column_one;
53     this-> N_p=Np_matrix(0,0);
54     // mu_x dimension 3x1
55     this->mu_x =( 1 / this->N_p )*
                        (matrix_src.transpose()*this->local_P.transpose()*
56     column_one);
57     // mu_y dimension 3x1
58     this->mu_y =(1 / N_p )*
                        (matrix_dst.transpose()*this->local_P*column_one);
59      // X_hat dimension Nx3
60      this->X_hat.resize(N,3);
61      X_hat = matrix_src - (column_one*this->mu_x.transpose());
62      // Y_hat dimension Nx3
63      this->Y_hat.resize(N,3);
64      Y_hat = matrix_dst - (column_one*this->mu_y.transpose());
65      // A matrix dim 3x3
66      A =X_hat.transpose()*
67      this->local_P.transpose()*this->Y_hat;
68      ///svd of A
69      Eigen::JacobiSVD<Eigen::MatrixXf> svd(this->A, Eigen::ComputeFullU |
                        Eigen::ComputeFullV);
70      Eigen::Matrix3f U=svd.matrixU();
71      Eigen::Matrix3f V_transpose=svd.matrixV().transpose();
72      Eigen::MatrixXf a=U*V_transpose;
73      float new_terms=a.determinant();
74      this->C(2,2)=a.determinant();
```

```
75      int det=(U*C*V_transpose).determinant();
76      Eigen::FullPivLU<Eigen::MatrixXf> lu_decomp(U*C*V_transpose);
77      int rank=lu_decomp.rank();
78      this->R=U*C*V_transpose;
79      //translation
80      this->t=this->mu_x-(this->R*this->mu_y);
81      //sigma square
82      Eigen::MatrixXf  diagonal=Eigen::MatrixXf::Zero(N,N);
83
84      Eigen::MatrixXf  partial = this->local_P.transpose()*column_one;
                         //N,1
85      for (int i = 0; i < N; i++) {
86            diagonal(i, i) = partial(i, 0);
87      }
88      this->sigma_square=(1/(this->N_p*3))*
89      ((this->X_hat.transpose()*diagonal*this->X_hat).trace()-
90      ((A.transpose()*this->R).trace()));
91      if(this->sigma_square<this->lowest_sigma and this->sigma_square!=0){
92           this->R_best=this->R;
93           this->t_best=this->t;
94           this->lowest_sigma=this->sigma_square;
95       }
96      counter++;
97      cout<<"Cycle"<<counter<<endl;
98     } while(this->sigma_square > max_thresh_sigma_square and counter <
                         max_iterations);
```

Chapter 5

Achieved Results

In this section in presented the results obtained using the two different cameras. The position of object represents the position of its centroid respect the camera, and the orientation is expressed respect the position of the key-points on the reference position of object.

5.1 Simulations using T265

5.1.1 Translation

In the following section is showed the results obtained using the tracking camera T265 during a translation along x and z axis. At the beginning is done the detection of the object and is extracted the key-points inside the region of interest (ROI). Then, it has been done only one time the matching between the features extracted from the left and right image,taken as reference, then during the execution of applications it has been done the matching between features extracted from left and right of run time image, and at the end between the matched features from run time and the matched features of reference image. In figure 5.1 is shown the matching between left and right run time images, and in the figure 5.2 the matching between the run time and reference image.

In this case the object does not change the orientation, so the expected Roll, Pitch, Yaw angles should be zeros, the position of object is expressed respect the camera and it represents the position of centroid of object. In the figure 5.3 is shown the position and the orientation of object, it is important specify that the position is expressed in *cm* and the angles in *degree*. In this case is possible to see the results obtained with **ICP** and with **CPD**, it is possible notice that CPD is more robust than ICP, in fact the results with ICP are really wrong, the Roll,Pitch,Yaw in the sequence x,y,z measured with ICP are 79.7°, 116.4°, 6.907°, instead with the CPD they are 0°, 0°, 0° how it is expected. Then it has been done the translation along x and z axis, the object is moved away from camera and is moved on the left, and it is possible see that the coordinate x decreases from 4.298 cm becomes 0.86 cm and the z increases from 19.418 cm to 22.148 cm, and about the orientation does not change, but only **CPD** has been able to reach

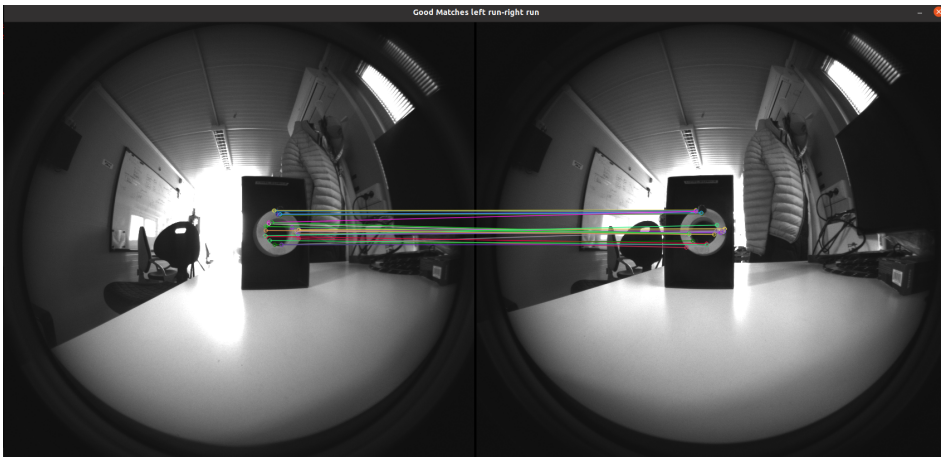


Figure 5.1: Shows matching between the two run time images of stereo camera (left and right)

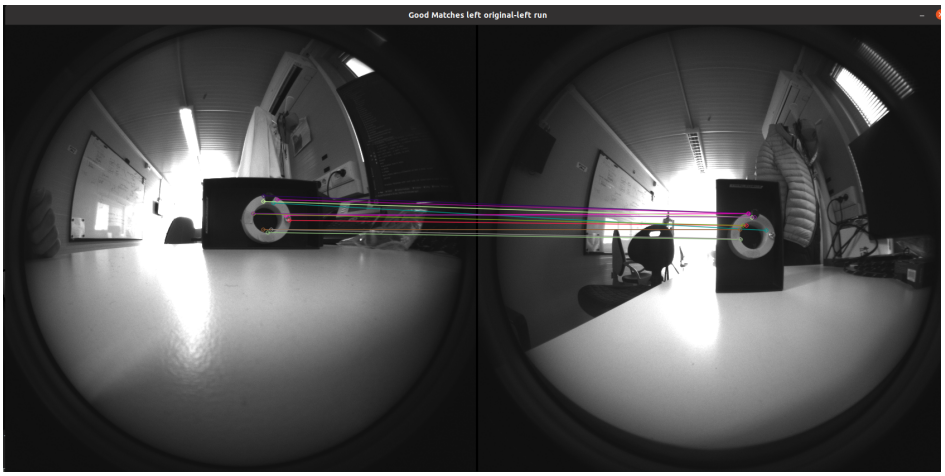


Figure 5.2: Shows the matching between the left reference image (on the left) and the left run time image (on the right)

solution (see figure 5.5), the reference system has the orientation showed in the figure 5.4.
In the figure 5.6 is shown the matching between reference and run time image after translation.

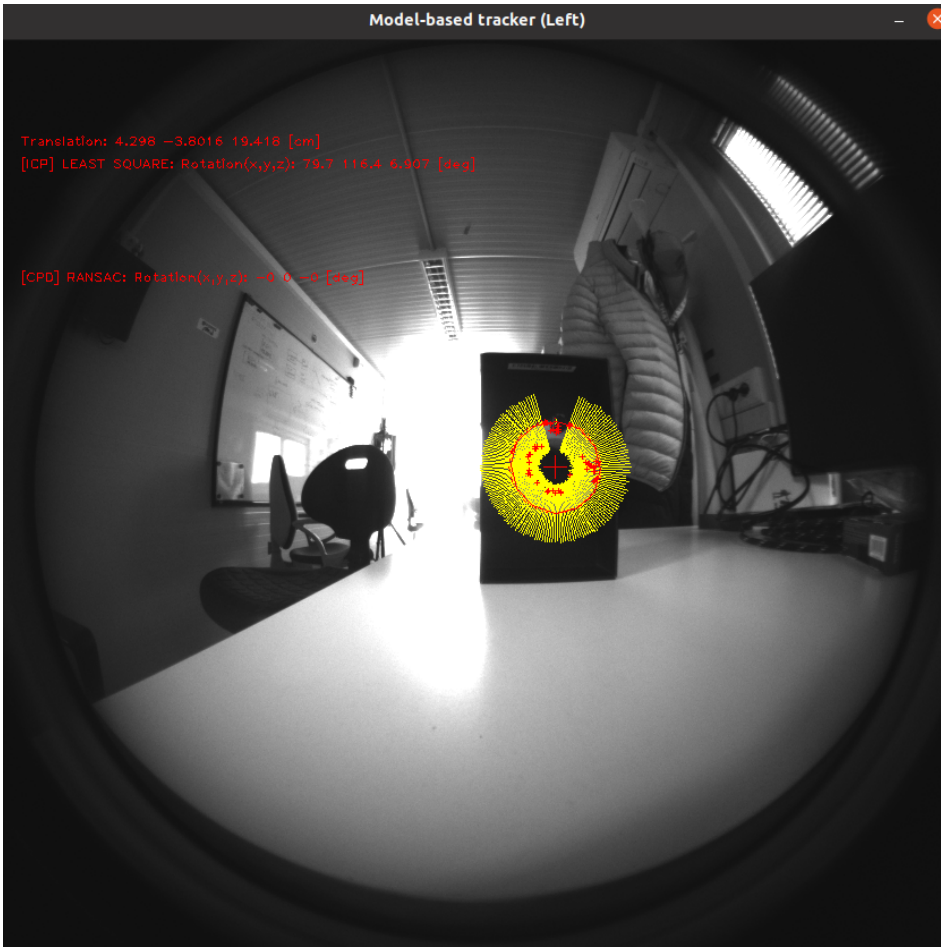


Figure 5.3: Pose estimation at the beginning

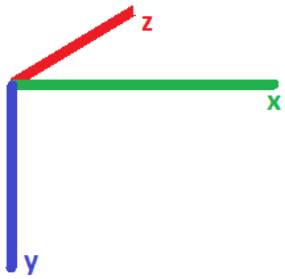


Figure 5.4: Reference system respect camera

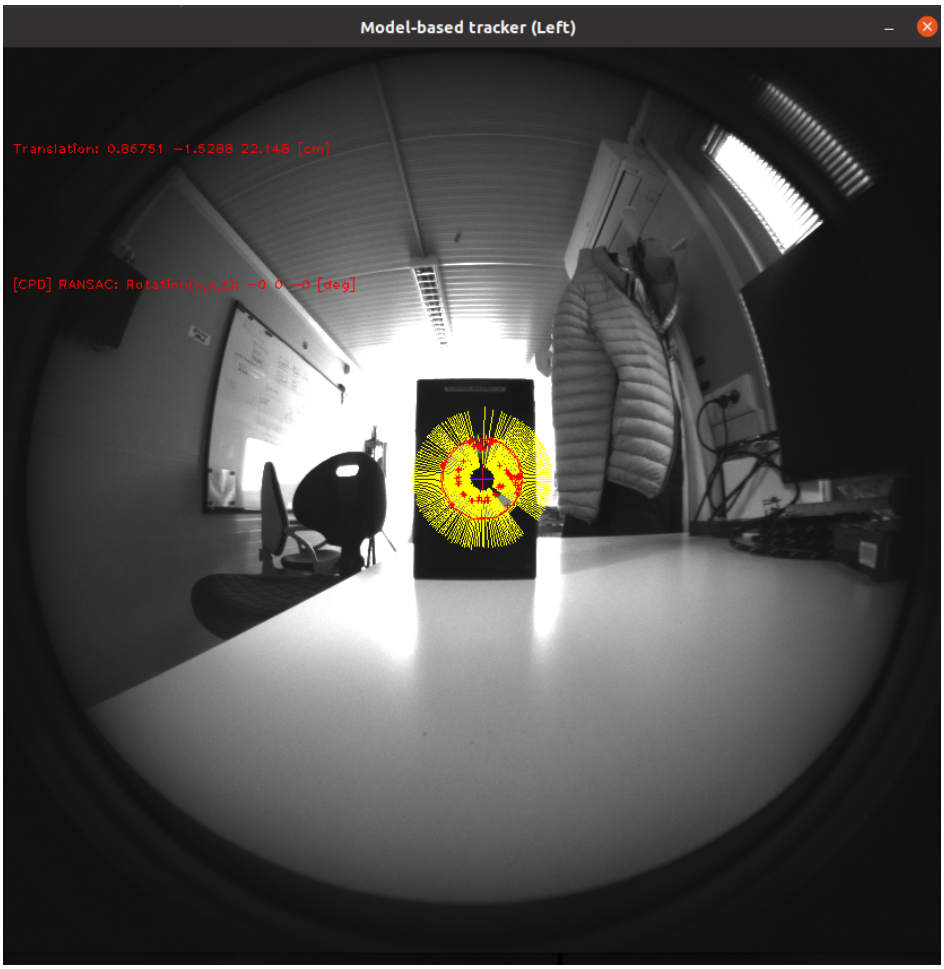


Figure 5.5: Pose estimation after translation along x and z axis

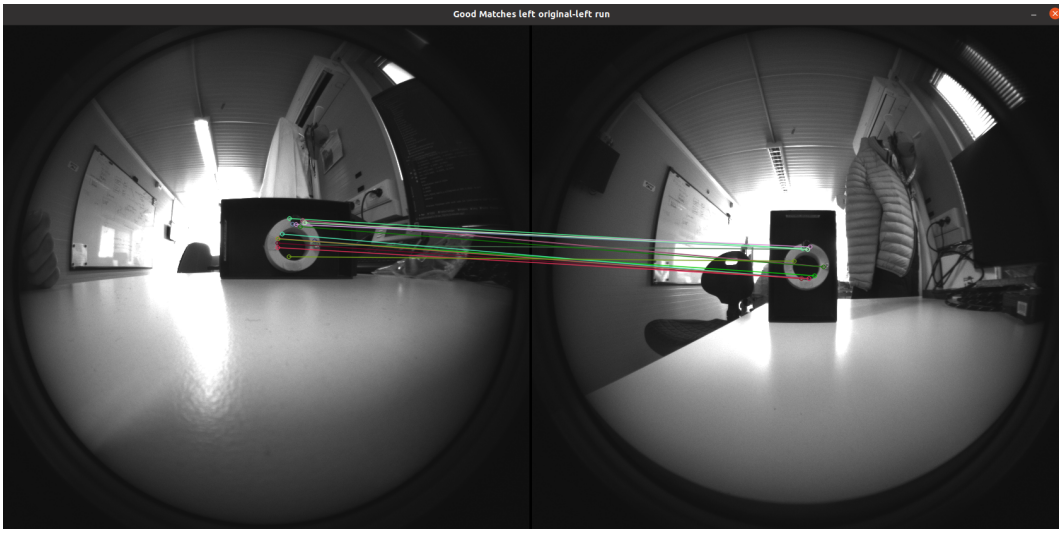


Figure 5.6: Matching between left reference image (on the left) and left run time image (on the right)

5.1.2 Rotation

In this section is showed the results obtained with a rotation of 90° around z axis. It measures the position of the object centroid respect camera and it is expressed in *cm*, the orientation is expressed with Roll,Pitch,Yaw around x,y,z axis (with this order) and they are expressed in *degree* and it represents the orientation of object respect the reference image. The process of detection, extraction and matching is equal to previous case. In fact, the extraction of features has been done only inside the region of interest, and the matching is done only one time between the left and right reference images, and then it has been done run time between left and right run time image and then between matched features run time and matched features reference. In the figure 5.7 is showed on the left the features on left reference image matched with the features on the left run time image. It is possible to see on run time image that the

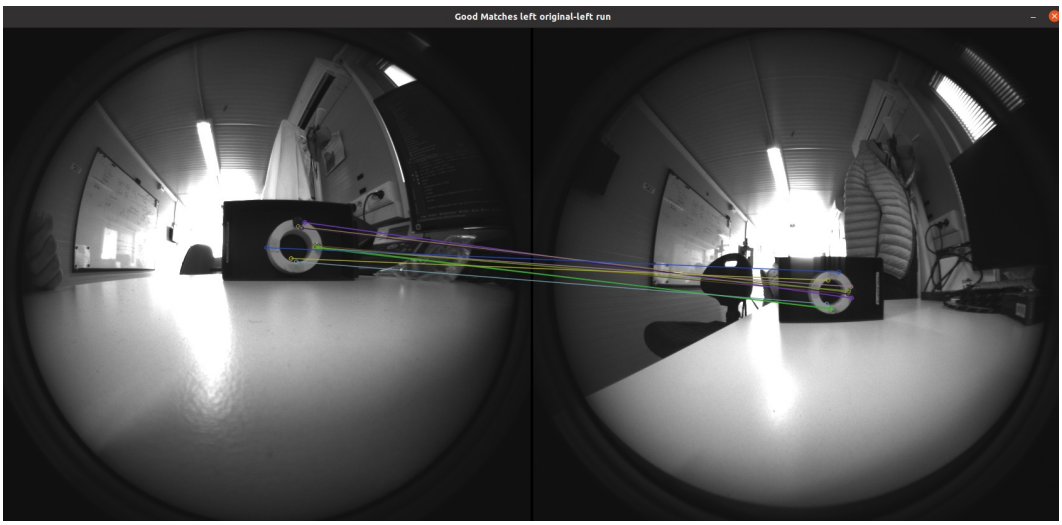


Figure 5.7: Matching between left reference (on the left) and left run time (on the right)

object is rotated by 90° around z axis, so the expected Roll,Pitch,Yaw should be 0° (around x), 0° (around y), +90° (around z), but using **ICP** it measures 20,55° around x, -29.12° around y, and +75.14° around z, so there is an error around 30° for all axis, instead using **CPD** there is a big error on the angle around y axis, in fact it measures 147.5° but should be 180° so there is an error around 32° , and the angle around z axis is 83.19°. So at the end CPD is more robust respect ICP, even if also CPD has errors. About the estimation of position is really precise in fact it measures 6.27 cm on x coordinate, 1.96 cm on y , 25.98 cm on z. (See figure 5.8). In the figure 5.9 is shown the rotation matrix obtained with **CPD** algorithm.

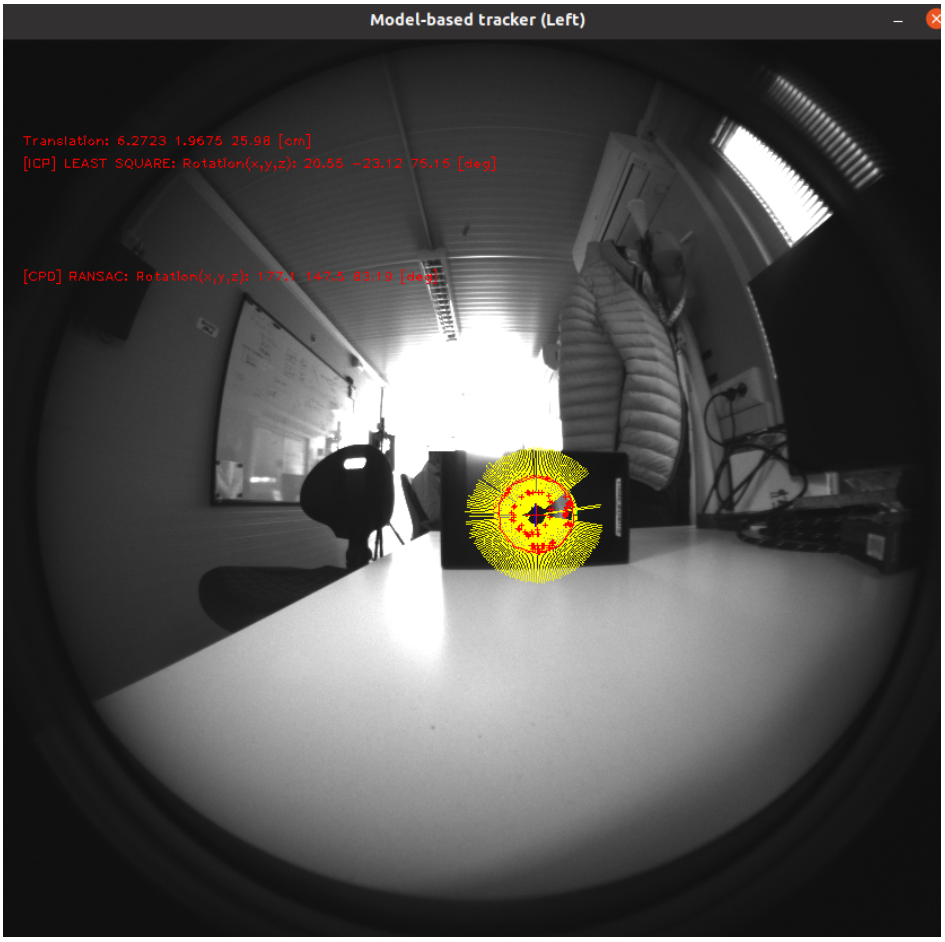


Figure 5.8: Pose estimation

```
[CPD Ransac] rotation:
   -0.1  0.8373  0.5376
-0.9885 -0.1452 0.04226
 0.1135 -0.5272  0.8422
[CPD Ransac] Best angles rotation from init to run:
177.1 147.5 83.19
```

Figure 5.9: Rotation matrix obtained with CPD algorithm

5.2 Simulations using D435

5.2.1 Translation

In this section is presented the results obtained with a translation of object along y axis. Firstly, the object is detected for the first time and starts the tracking (see figure 5.10). Then, it is extracted the key-points and they are matched with the key-points

Figure 5.10: On the left shows the detection of object and on the right the first tracking of object

on reference image (5.11), and it is obtained the results shown in the figure 5.12, where Roll,Pitch,Yaw around x,y,z are zeros degree, how it is expected, using **CPD** algorithm and with Least square **ICP**, the angles are not really accurate, in fact they are 84.6° around x, -126° around y, 177° around z axis, so there is an error of 96° for the angles around x,and 50° for angles around y. The initial position of centroid of object respect the camera is 0.04 m on x , -0.008 m on y and 0.49 m on z (this last is the distance from object and camera).

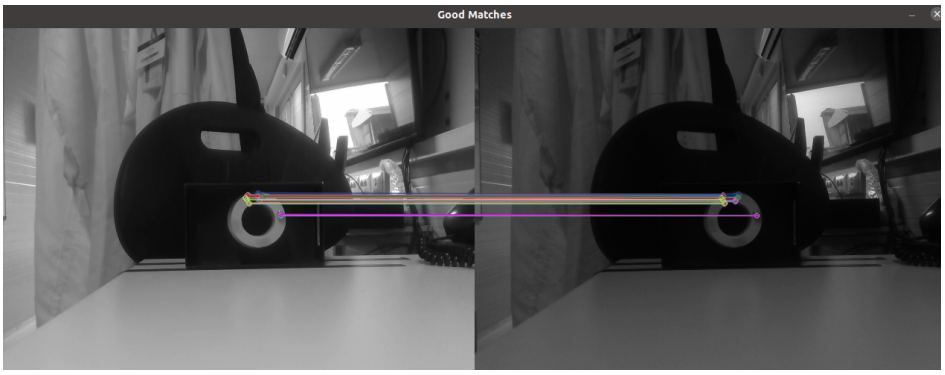


Figure 5.11: Matching between run time image (on left) and reference image(on right)

Then, the object is moved upward, this is visible in the figure 5.13 and the new matching is shown in figure 5.14 and the pose is shown in the figure 5.15. About the pose estimation, it is possible see that the position of centroid on coordinate x is not changed, the coordinate y changes it becomes -0.15 m and about the distance from camera it becomes 0.49 m. The orientation does not change, and with CPD the Roll,Pitch

Figure 5.12: Initial pose estimation

and Yaw around x,y,z axis are zeros. It is important notices that in the Fig. 5.15 there is not the Roll,Pitch,Yaw computed by ICP, this because the algorithm was not able to achieve a good results with this configuration. From these results is possible understand the reference system respect the camera, in fact with a translation upward the coordinate y decreases,and if the object goes away from camera the coordinate z increases, in the figure 5.4 is shown the reference system.

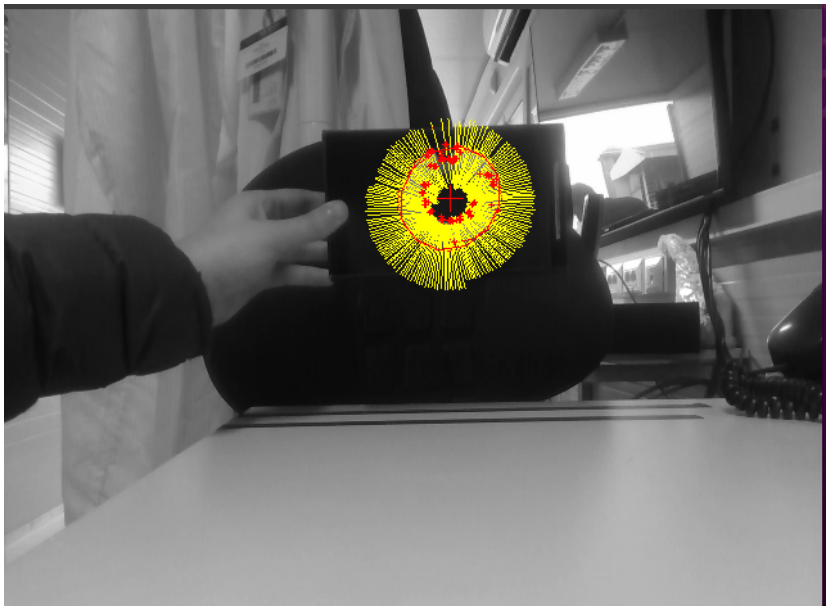


Figure 5.13: Tracking during the movements

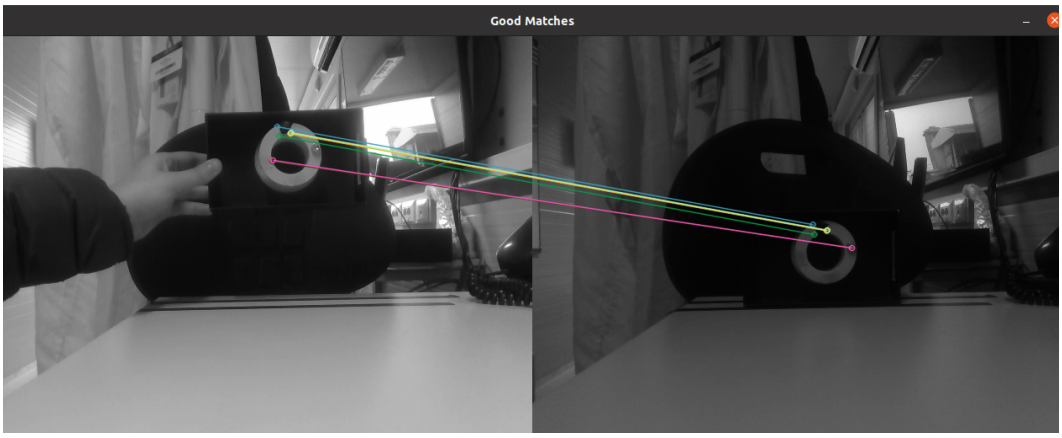


Figure 5.14: Matching after translation

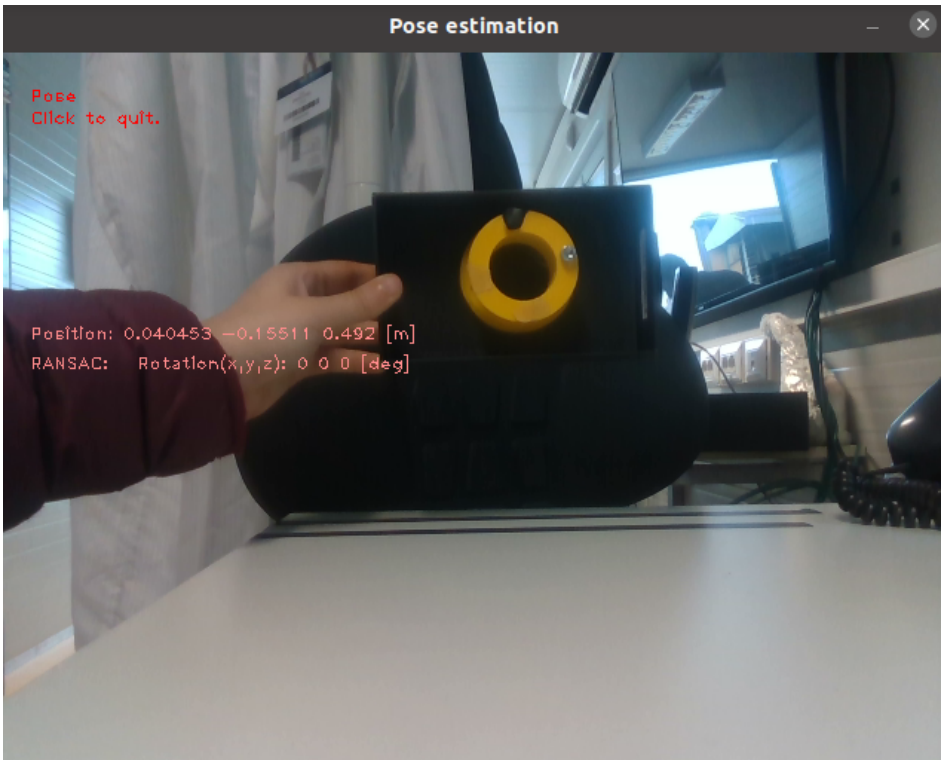


Figure 5.15: Pose estimation after translation along y

5.2.2 Rotation

In this subsection is shown the results obtained during a rotation of object of 90° around z axis. Firstly, is detected the object (5.16).

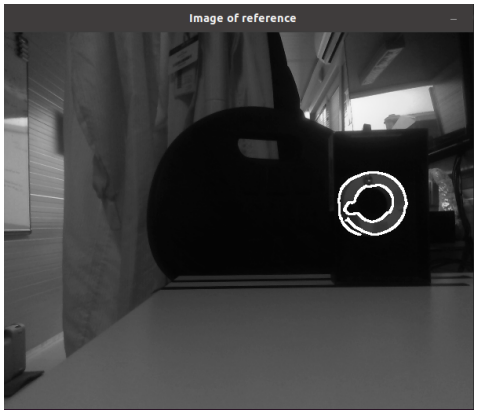


Figure 5.16: The object is detect for the first time

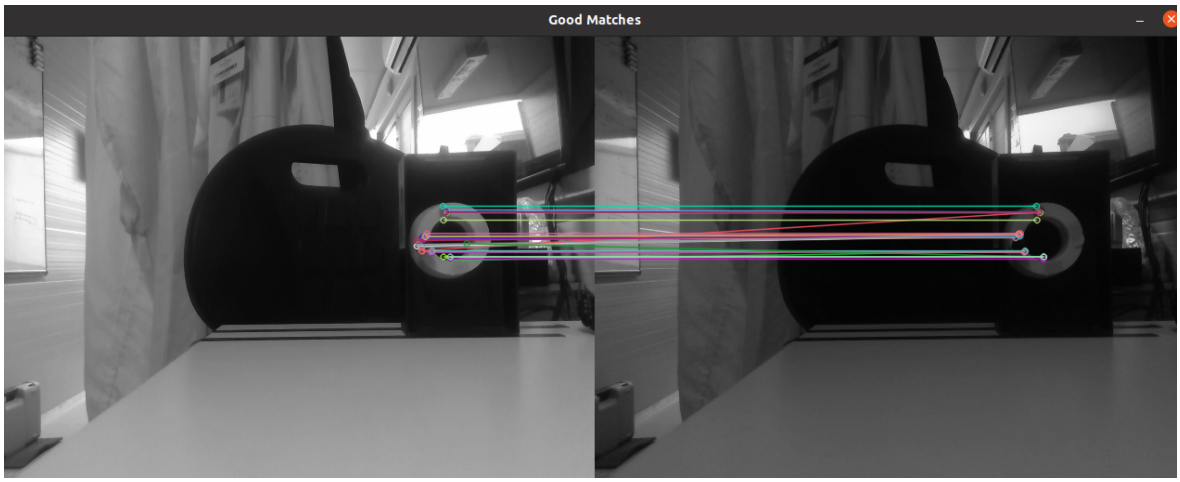


Figure 5.17: On the left is the run time image, on the right reference image (case where the run time and reference pose are equal)

Then, is done the matching between the key-points on run time image and reference image (see figure 5.17), at the beginning the object has the same position and the same orientation of the reference pose, so the expected results for the orientation expressed with Roll,Pitch,Yaw (with x,y,z order) should be equal to zero (angles are expressed in degree), in the figure 5.18 is shown the position of object centroid respect camera, the distance is expressed in meters, about the angles in the figure is shown two different measurements, the second line shows the computation of Roll,Pitch,Yaw using the **CPD** with **Ransac**, it is very accurate there are a difference of 3° on rotation around y axis, the measured angles are 1.13° around x, 3.268° around y and -0.361° around z. Also in this case is not visible the results computed by ICP because the algorithm was not able to achieve good results with this configuration. In the 5.19 is possible to see the mean errors of projection around each axes, where the highest value of error is around

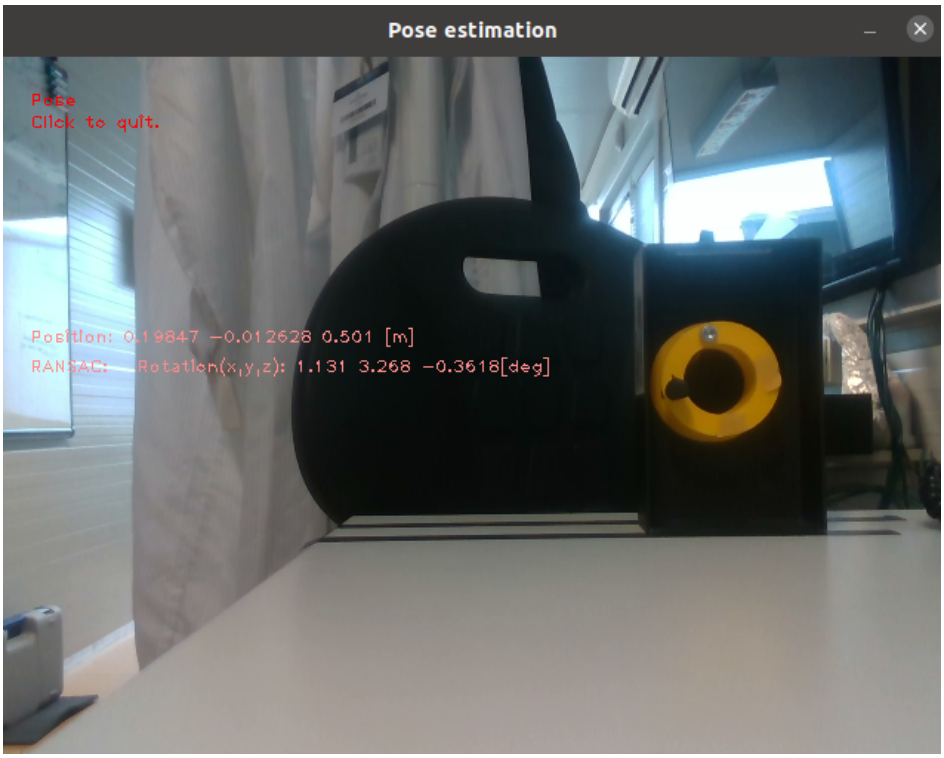


Figure 5.18: Initial pose estimation respect reference pose

y axis, it is 9 mm, respect the other that are around 1 mm. It is possible to see also the rotation matrix that is really similar to identity matrix, how it is expected, About the *best translation* shows the translations of the key-points from reference position to actual position, the translation is around 1 cm on each axis. Then is done the rotation

```
 Mean projection error on axis(x,y,z) with CPD[m]:
 0.000185397 -0.00966534 0.00140456

Best traslation with CPD algorithm:
-0.0283373
 0.0116223
 0.0141079

Best rotation with CPD algorithm:
   0.998354   0.0063048   0.0570111
-0.00518876    0.999793  -0.0197063
 -0.0571239    0.019378     0.99818

(CPD) Best angles rotation from init to run:
 1.131 3.26827 -0.361842

Centroid on reference[m]:
[0.188238, -0.0306881, 0.504]

Centroid run time[m]:
[0.198474, -0.0126282, 0.501]
```

Figure 5.19: It is shown the mean errors of projection, rotation matrix

of object, in the figure 5.20 is shown the matching between the key-points extracted on run time image (on the left) after a rotation of 90° of the object, and on the right reference pose. The results are shown in the figure 5.21, in this case is possible to see on the first line the position of centroid of object respect the camera(in meters), it

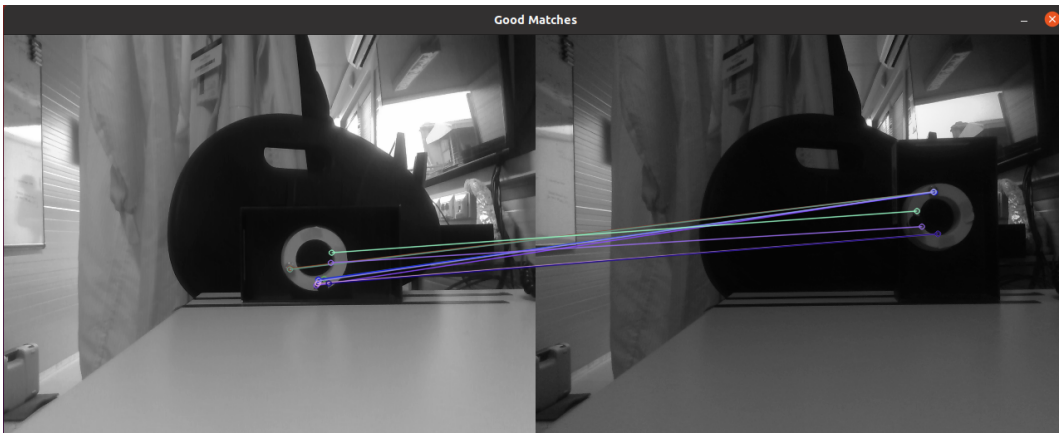


Figure 5.20: Matching after rotation of 90° (on the left the new pose of object and on the right the reference pose)

does not change the z coordinate it remains 0.50 m, this because the object remain at the same distance from the camera, however the x coordinate from 0.19 m becomes 0.07 m because the object centroid is moved towards left, on the second line the result computed using **CPD** with **Ransac**, and on the third line shows the angles computed with the **Least square ICP** with **Ransac**. It is possible see that the CPD is more accurate respect the ICP, this is due to the presence of outliers (how is possible to see in the figure 5.20), in fact CPD is able to take account of outliers, the least square ICP is not able, even if it is wrapped with Ransac. Using CPD the rotation around axis x

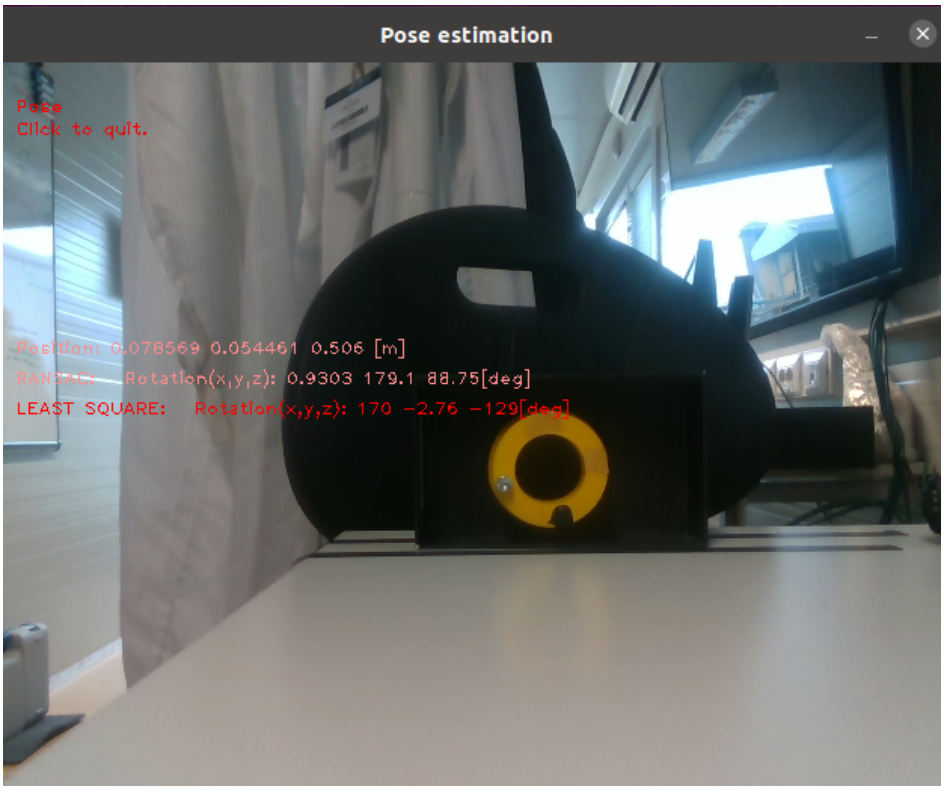


Figure 5.21: Pose estimation, respect reference pose, after rotation

is 0.93°, around y 179.1° and around z is 88.75°, these are Roll,Pitch,Yaw around x,y,z

and the angles expressed in this way can be not intuitive,so in the figure 5.22 is shown the rotation matrix, where is possible see that there is a rotation around z axis of an angle near 90°.

```
 Mean projection error on axis(x,y,z) with CPD[m]:
 -0.00602347 -0.0269732 -0.0106283

Best traslation with CPD algorithm:
 0.062365
-0.098325
  0.01575

Best rotation with CPD algorithm:
-0.0217411    0.99965   0.015065
 -0.999637  0.0214961  0.0162339
 0.0159043  0.0154125   0.999755

(CPD) Best angles rotation from init to run:
 0.930285 179.137 88.7541

Centroid on reference[m]:
[0.214238, -0.0544898, 0.509]

Centroid run time[m]:
[0.0785694, 0.054461, 0.506]
```

Figure 5.22: Output obtained after rotation of 90°

5.3 Comparison

As seen in the previous sections, the results about orientation obtained by depth camera D435 is more accurate respect the results obtained by T265, the reason is because the depth camera is used for the pose estimation of object and the T265 works very well for the tracking or for SLAM applications. Another reason is that the numbers of 3D points, used to estimate the transformation matrix, are more respect the 3D points extracted by T265, in fact the algorithms, used for its estimation (**CPD,ICP,Homography**) work better if the sets of 3D points have more numbers of points. Using the T265, where in order to find the 3D points on run time image, that correspond to 3D points on reference image, it is necessary do three matching so at the end of process in more difficult to find a lot of correspondent points. About the estimation of position the results obtained by T265 are better respect the results obtained by D435, this because the triangulation is done on specific points and it has been done using OpenCV library.
About the algorithms implemented for the estimation of rotation matrix, the best results has been obtained using CPD[section 4.7.3], because it is a probabilistic method and takes account of numbers of outliers. The least square ICP [section 4.7.2] works properly when there are not outliers, so it is able to obtain a accurate results only in ideal case, even if the ICP is wrapped with Ransac, and in this case it reaches a good results only with one or maximum two outliers. The worst algorithm is the Homography estimation [section 4.7.1], it does not work for 3D points, in fact it is the best to use only with 2D points. Using this method for the 3D case, usually system goes to singularity so the algorithm is not able to compute the rotation matrix.
It is possible summarize these comparisons in two tables. The table 5.1 compares the two cameras and the table 5.2 summarize the pro e cons between algorithms used to estimate orientation.

Camera	**Estimation of position**	**Estimation of orientation**
T265	best estimation	bad estimation, because the numbers of points are less
D435	good estimation	good estimation

Table 5.1: Summary results obtained by the two cameras

Algorithm	**Estimation of orientation**
Homography 3D	it does not work properly for 3D
ICP	it works properly only without outliers
CPD	it works properly also with outliers

Table 5.2: Summary pro and cons algorithms

Chapter 6

Conclusions

6.1 Conclusion and future work

In this work it has been treated the methods used for the 6D pose estimation. It has been shown two different approaches one is the markers based method and the second is the model based method.

The markers based method has been applied using three different kind of markers, with the *myMarkers* Library, that used unconventional kind of markers, has been reached good results for the estimation of position, when markers are positioned at small distance from camera, even if about the orientation is not able to estimate a good rotation if the markers are rotated with high speed. The second application is done using *QrCode* Library, the achieved results have shown that this kind of markers works property even if they are far from camera, but about the estimation of orientation it works only if the markers are rotated slowly. The last Library used for the markers based approach is *Aruco* markers, it shows a really accurate results not only for the estimation of position, also when markers are far from camera, but also the estimation of orientation are really good, it is able to follow also a fast rotations and movements of markers.

About the second approach, the model based method, it has been done using two different cameras the first using a stereo fisheye camera (T265) and the second using a depth camera (D435). The results obtained by the T265 camera are really good for the estimation of position of object, about the orientation it is not work really good as the depth camera, the reasons are two , the first is that this camera is a tracking camera, so for this kind of application is not really good, but the second reason is that in order to extract the 3D points it has been done three matching, so at the end of process there will be few 3D points, and the algorithms of point set registration work very well if there are a lot of points. About the algorithms implemented for the estimation of orientation of object the worst algorithm is the *homography estimation*, it does not work properly for 3D case, in fact it goes to singularity and it is not able compute

a rotation matrix, the second algorithm *ICP*, it is a good algorithm only in the case when there are not outliers, but it is not a real case, for this reason in order to increase robustness it was wrapped with a Ransac, this has been improved the algorithm, but it is able to reach a good results only with the presence of one/two outliers. Finally, the last algorithm is *CPD*, with this algorithm it has been obtained the best results, this is because it is a probabilistic method and it takes account of presence of outliers, in order to increase robustness it is wrapped with Ransac, in this way even if there are five/six outliers respect a total numbers of fifteen points, it is able to obtain a good results.
However, this last algorithm it has a limitation, in fact it requires to choose the values of some parameters a priori, like the ratio outliers that represents the numbers of noise on scene.
The two different methods has pro and cons, in fact using a markers approach the accuracy of pose estimation are higher, but it has the limitations that the markers shall be positioned around the object. This limitations is not present using the model based approach, and for space applications it is really important. In fact this method can be applied to estimate a pose of an object or a satellite of which we know only the shape, which is not possible attached markers around it.
The next step will be:

- introduce an deep learning algorithm to improve feature extraction.
- improve the robustness to outliers.
- increases the image of reference in order to detect also other face of object.
- perform a demonstration in closed loop, demonstrating visual servoing control of a Robotic Arm with wrist mounted arm.

Bibliography

[1] Papert and Seymour. "The Summer Vision Project". In: *MIT AI Memos* (1966).

[2] Margaret Ann Boden. "Mind as Machine: A History of Cognitive Science". In: *MIT AI Memos* (2006), p.781.

[3] Dharini Raghavan and Bhavana B Rao. "Computer Vision for Space Exploration". In: *International journal of engineering research technology (IJERT)* Volume 09 (2020).

[4] Filippo Bergamasco and Andrea Albarelli and Andrea Torsello. "Image-Space Marker Detection and Recognition using Projective Invariants". In: (2012).

[5] L. Gatrell, W. Hoff, and C. Sklair. "Robust image features:Concentric contrasting circles and their image extraction". In: *Proc. of Cooperative Intelligent Robotics in Space,* (1991).

[6] Youngkwan Cho, J. Lee, and U. Neumann. "A Multi-ring Color Fiducial System and A Rule-Based Detection Method for Scalable Fiducial-tracking Augmented Reality". In: 1998.

[7] David Claus and Andrew W. Fitzgibbon. "Reliable Automatic Calibration of a Marker-based Position Tracking System". In: *Proceedings of the IEEE Workshop on Applications of Computer Vision*. Jan. 2005, pp. 300–305. URL: www.robots.ox.ac.uk/%5C~%7B%7Ddclaus/publications/claus05f%20idpose.html.

[8] Diego Cesar et al. "An evaluation of artificial fiducial markers in underwater environments". In: May 2015, p. 6. DOI: 10.1109/OCEANS-Genova.2015.7271491.

[9] RasterGrid Kft. *Efficient Gaussian blur with linear sampling*. URL: https://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/.

[10] R. Fisher and S. Perkins and A. Walker and E. Wolfart. "Gaussian Smoothing". In: (2003).

[11] URL: https://computergraphics.stackexchange.com/.

[12] Fabio Nelli. *OpenCV Python – Canny Edge Detection*. URL: https://www.meccanismocomplesso.org/en/opencv-python-%20canny-edge-detection-2/.

[13] URL: https://docs.opencv.org/.

[14] Carlo Tomasi. *Vector Representation of Rotations*. URL: https://courses.cs.duke.edu/.

[15] Nasim Hajari,* Gabriel Lugo Bustillo, Harsh Sharma, and Irene Cheng. "Marker-Less 3d Object Recognition and 6d Pose Estimation for Homogeneous Textureless Objects: An RGB-D Approach". In: (2020).

[16] Brachmann ,E. and Michel ,F. and Krull, A. and Ying Yang, M.and Gumhold, S. and Rother, C. "Uncertainty-driven 6d pose estimation of objects and scenes from a single rgb image". In: *IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 3364–3372.

[17] Henning Tjaden, Ulrich Schwanecke, and Elmar Schomer. "Real-Time Monocular Pose Estimation of 3D Objects Using Temporally Consistent Local Color Histograms". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[18] C.Gupta S. and Girshick R. and Arbeláez P. and Malik J. "Learning rich features from RGB-D images for object detection and segmentation". In: *European Conference on Computer Vision* (2014), pp. 345–360.

[19] Simon M., Milz S., Amende K., Gross H.M. "Complex-YOLO: An Euler-Region-Proposal for Real-Time 3D Object Detection on Point Clouds". In: (Oct 2018).

[20] E. Marchand, F. Spindler, and F. Chaumette. "ViSP for visual servoing: a generic software platform with a wide class of robot control skills". In: *IEEE Robotics and Automation Magazine* 12.4 (Dec. 2005), pp. 40–52.

[21] K.S. Arun, T.S. Huang, and Steven Blostein. "Least-squares fitting of two 3-D point sets. IEEE T Pattern Anal". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* PAMI-9 (Oct. 1987), pp. 698–700. DOI: 10.1109/TPAMI.1987.4767965.

[22] Andriy Myronenko and Xubo Song. "Point Set Registration: Coherent Point Drift". In: *IEEE transactions on pattern analysis and machine intelligence* 32 (Dec. 2010), pp. 2262–75. DOI: 10.1109/TPAMI.2010.46.

[23] C. M. Bishop. "Neural Networks for Pattern Recognition". In: *Oxford University Press* (1995).

Acknowledgements

Vorrei ringraziare il mio relatore Marcello Chiaberge e il suo team per la disponibilità e per avermi dato l'opportunità di svolgere questo lavoro di tesi.

Ringrazio il mio supervisore Andrea Merlo per avermi permesso di poter lavorare su un progetto interessante e coinvolgente, e per avermi dato la possibilità di svolgere questo lavoro di tesi in un luogo interessante e dinamico, che mi ha permesso di mettermi in gioco e fare esperienza.

Ringrazio Ciro,Genny,Alessandro,Paolo,Marco,Patrick,Fabio e tutti i frequentatori della palazzina 77 per avermi accolta, supportata, aiutata durante questi mesi.

Ringrazio i miei genitori per avermi sempre sostenuta, appoggiando ogni mia decisione e per aver permesso di portare a termine il mio percorso universitario.

Ringrazio mia sorella per avermi incoraggiata dall'inizio del mio percorso universitario.

Ringrazio i miei amici Matteo e Nicolò per avermi aiutata e spronata.

Ringrazio Roberto per essere stato sempre presente e aver sopportato i momenti di sfogo.

Ringrazio Vita per aver condiviso con me questi anni universitari, condividendo con me gioie, ma anche momenti difficili.