Politecnico di Torino

Master of Science in Mechatronic Engineering

Master of Science Degree Thesis

Development and testing of an industrial vision application for robot guidance using OpenCV library

Supervisor:
Prof. Marcello Chiaberge

Candidate:
Enrico Mollo

December 2023

Abstract

The purpose of this thesis work is the development of an open-source industrial vision application for robot guidance, implemented in Python language using Open CV library. The developed program aims to be a free-of-license alternative to the vision software of the industrial machine Supata®, entirely designed, realized, and programmed by E.P.F. Elettrotecnica. It consists of a vibrating platform (the Supata® itself), a camera with a light source, and a robot, to compose a smart feeder system that singularizes parts randomly loaded into it and prepares them for following stations.

This work covers all the development phases of this application, from the initial camera configuration to the achievement of the final output: detect pieces that can be picked, their orientation and grip point's cartesian coordinates, and, finally, the density distribution on the platform. The procedure starts with the calibration of the camera, necessary to eliminate the lens distortion effects, followed by the definition of the correlation between the 2D pixel coordinates of the image and the 3D millimetres coordinates of the robot.

Once the camera is ready and the input image of the platform is acquired, the main program operates on two sides: a setup side, performed only the first time a new piece is considered, and a processing side, performed every time a new image is acquired.
The setup side includes the definition of the master image and all the concerning parameters and features. On the processing side, after an initial selection based on the shape of the objects, the program compares every piece with the master, computing its orientation and the coordinates of the grip point. Then, after a final control to check the presence of obstacles in the grip area, the final list of pickable pieces is returned, as well as their orientation and grip point coordinates.
Besides the coordinate computation, it also gives an information about the distribution of the remaining pieces on the vibrating platform.

The comparison phase between a generic piece and the master is the core of this project and the most challenging aspect of the whole work, since it is the key to make this program applicable for every kind of piece, no matter the shape, the material, or the colour. To accomplish this task, a fundamental technique in computer vision known as feature matching is adopted, where the features of two images are detected, described, and matched by specific algorithms, implemented through dedicated OpenCV functions: in particular, the algorithms considered for this project are SIFT and ORB for the detection and the description of the features, and Brute-Force and FLANN for the matching.

Finally, to test the program, a set of experiments has been prepared in which six different combinations of detector/descriptor and matcher algorithms have been applied on four pieces with very different physical characteristics: for each test it has been considered as quality evaluation criteria the number of well-matched and wrong-matched pieces, the computational time, and the precision with respect to the currently implemented industrial vision software on Supata®, based on Cognex libraries. The results of the tests show how the combination of ORB as detector and descriptor algorithm with FLANN plus homography as matcher algorithm has the best performance for every quality criterion, revealing itself as a promising starting point for future improvements.

Acknowledgments

I would like to thank the staff of E.P.F. Elettrotecnica for the opportunity they gave me and for introducing me to the world of robotics. A special thanks to Giulio Pugliese for supporting me along all the developing of this thesis work and to Professor Marcello Chiaberge for being my supervisor.
My parents, my girlfriend, and my friends already know how thankful I am for the love and support they showed me along the way.

List of Contents

Chapter 1: introduction

1.1 Work presentation

In recent years, with the transition to smart factories and Industry 4.0, machine vision has become one of the most important research fields for industrial automation, since it gives to industrial equipment the ability to 'see' the surrounding world and make real-time fast decisions based on this vision.
Robots, in particular, when provided with industrial vision can understand and recognize shapes, calculate volumes, identify objects and fulfil much more complex contact-free tasks as measuring, improving the product quality, the overall systems efficiency and the operator's health and safety, reducing labour costs and, in general, optimizing manufacturing and logistics.

The aim of this thesis work is the development of an industrial vision application for robot guidance for the E.P.F. Elettrotecnica machine Supata®, consisting of a vibrating platform (the Supata® itself), a camera with a light source, and a robot, to compose a smart feeder system that singularizes parts randomly loaded into it and prepares them for following stations.
This application, implemented in Python language using OpenCV library, aims to be a valid alternative, in terms of performances, to the software currently implemented on Supata® based on Cognex libraries, but completely open-source and free-of-license.

This work covers all the software development phases, from the initial camera configuration to the detection of the pieces that can be picked, their orientation, the cartesian coordinates of grip point, and, finally, the density distribution on the platform.

1.2 Concerning Supata®

Supata® (Figure 1.1) is an industrial machine completely designed, programmed, and realized by E.P.F. Elettrotecnica, including mechanical, electrical and software. It is a digital, intelligent, flexible feeding module for industrial automation sector, equipped with a high-precision customizable integrated vision system.

It consists of a manipulator able to pick pieces from a vibrating platform: the robot can recognize the well-posed pieces and when all of them have been moved away, new pieces are loaded on the platform that randomly re-distributes the new pieces with a vibrating motion regulated by two separate motors, one on the left and one on the right side of the platform. Every time the vibrating platform re-distributes the pieces, the camera takes a picture, the system recognizes the well-posed pieces and returns the grip point coordinates and the piece rotation angle to the manipulator that proceeds with the grip operation.

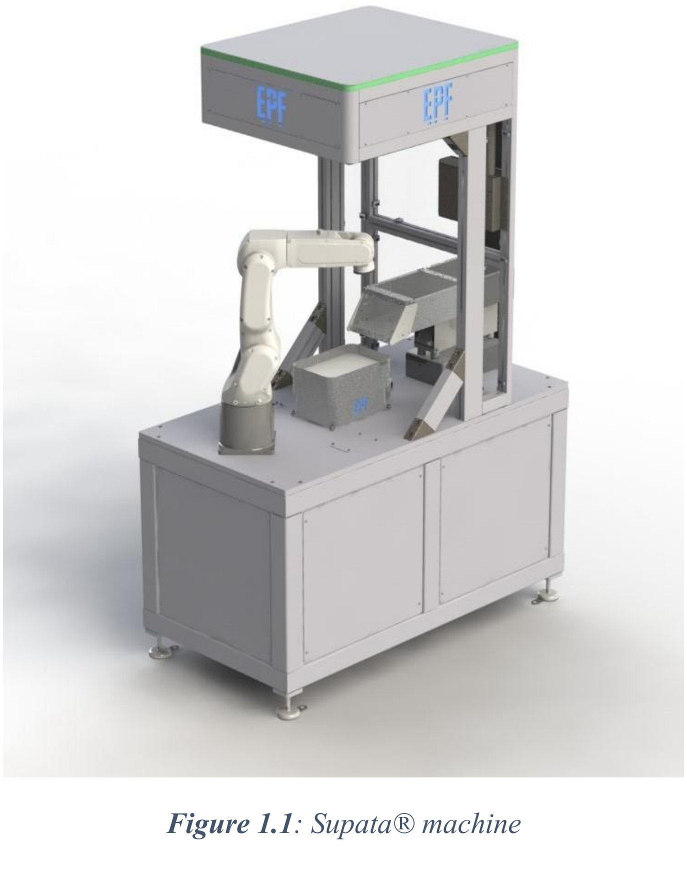


***Figure 1.1**: Supata® machine*

As the company website [1] states, Supata® is designed and suitable for:

- Be simply integrated into existing lines.
- Produce small batches with frequent production changes.
- Be configurable according to different applications.
- Manage components with different sizes and geometries.
- Replace rigid and unreliable vibratory feeders.
- Create a simple system, always ready for new products, with a single interface.
- Obtain fast and regular cycle times.
- Improve efficiency and quality of production processes, eliminating costly rework.
- Ensure traceability throughout the production chain.
- Gain a competitive advantage.

Also, by adopting A.I., the productivity can be improved by 17-20% by optimizing quality. Errors can be eliminated reducing costly rework and the overall efficiency is maximized, while costs are reduced.

Nowadays Supata® system is used for different fields of industrial production as automotive, food and beverage, medical, gadgets, household appliances, household goods and all production sectors where the use of mechanical vibrators are required.

1.3 Concerning OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library, supporting C++, Python, Java and MATLAB and compatible with Windows, Linux, Android and Mac OS.
The library has more than 2500 optimized algorithms for computer vision that can be used for simple operations of image processing as geometric transformation, thresholding, histogram equalization but also for more specific applications as edges and contours detection, template matching, image segmentation and to solve very complex tasks as

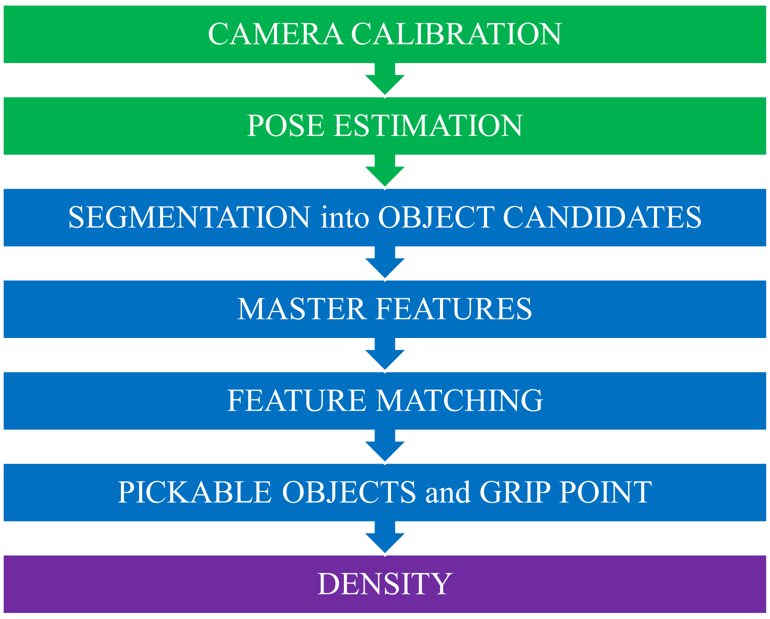
feature matching. That's why OpenCV algorithms are widely used in companies, research groups and by governmental bodies to detect and recognize faces, track moving objects, find similar images in a database and so on, with more than 47 thousand people of user community and over 18 million of estimated downloads all over the world.

Since this project concerns a computer vision problem, OpenCV library is the core element to find the solution, and the main OpenCV functions implemented in the code are going to be discussed and explained in detail in the following chapters.

Figure 1.2*: OpenCV logo [2]*

1.4 Solution approach

After this introduction, each chapter will deal with a different part of the code development, while the last chapters are dedicated to the test phase and the conclusions. To give a clearer idea of how this project has been organized, a general scheme of the code development is reported below, and it is subdivided into its main steps, that are briefly resumed according to the chapters organization order.



1) **Camera calibration**. This step is quite common in computer vision application since it is necessary to eliminate the lens distortion effect. The program receives as input several images of a special chessboard-like pattern called ChArUco board, placed in different positions and returns two parameters, characteristics of the lens used to undistort any input image.

2) **Pose estimation**. Once the undistorted image is acquired, it is fundamental to find a correlation between the 2D image world in pixels and the 3D real world in millimetres. The program places the global reference frame (X_w Y_w Z_w) and returns all the parameters necessary to find the pixel-to-mm transformation. Once all the parameters are known, the correlation is obtained by inverting the pinhole camera model equation.

3) **Segmentation into candidate objects**. First, the user must indicate the grip constraints, to avoid the collision between the robot and the platform borders, and the master image. With a very important step known as segmentation, based on the thresholding operation, the program obtains the blob (silhouette) of every piece in the platform, where the segmentation parameters are chosen by the user. The program also computes, for each piece, the blob centroid and the area, that is compared with the master blob area: if it is included in a certain tolerance range, the corresponding piece is considered as candidate object.

4) **Master features**. All the operation concerning the master are performed only the first time that a new kind of piece is considered, while all the other operations are performed every time a new image is acquired. In this phase the user must impose four important features operating directly on the master image: the collision avoidance area, the master grip point, the master orientation, and the keypoints area. The collision avoidance area is the space that must be obstacle-free to consent the grip, and it is defined by tracing two polygons around the piece. The master grip point is simply indicated by a dot, while the master orientation by an arrow. Finally, the keypoints area is the area of interest for the feature matching: it is defined as a polygon and every keypoint outside the polygon is not considered for the matching.

5) **Feature matching**. This is the core of the whole project: each candidate object is separated in a target image that is compared to the master by their features (keypoints) which are detected, described, and matched by specific algorithms. The detectors/descriptors chosen for this project are ORB and SIFT, while the matchers are Brute-Force, FLANN, and FLANN + homography. Differently from segmentation, this procedure leads to a comparison not based on the shape of the pieces, but on their features, independently from their pose: this makes the program applicable for every piece, no matter the shape, the material, or the

colour, discarding wrong-posed pieces and obtaining the rotation angle of the target pieces with respect to the master (orientation).

6) **Pickable objects and grip point**. Once the rotation is obtained, for each target piece the software computes the coordinates in pixels of the grip points, and the presence of obstacles in the collision avoidance area is checked through the Canny edge detection method. If a target piece passes all these controls, it is classified as pickable, and the coordinates of its grip point are converted from pixels to millimetres and provided to the user along with the piece rotation.

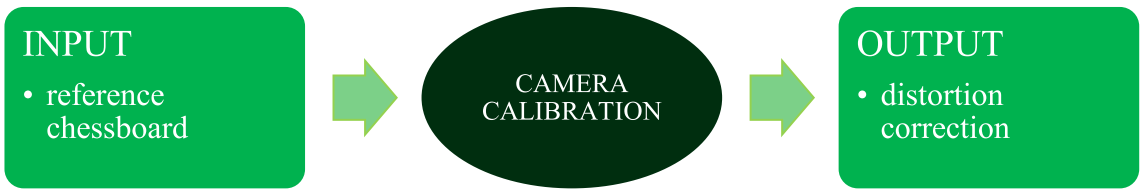
7) **Density**. Once all the pickable pieces have been moved away, it is possible to obtain an information about how the remaining pieces are distributed on the platform by applying again the thresholding. This information is needed to decide if new pieces must be loaded on the plane, or they must be vibrated instead.

Chapter 2: camera calibration

2.1 Introduction

Camera calibration is the first step for any computer vision project work, since we can say that the 'eyes' of industrial robots consist in one or more cameras and their lenses, optical devices characterized by undesired effects as distortion.
Therefore, the main goal of camera calibration is to correct those effects, obtaining the necessary sets of parameters to correct the camera view, from which it is possible to acquire the undistorted images to be processed in the following part of the project.

The base principle of the calibration procedure consists in the realization of a chessboard-like pattern with well-known square sizes. From several pictures of this chessboard taken by the camera, the program returns all the camera parameters to correct the original images and therefore to obtain their undistorted version.
For a matter of robustness, it has not been used a simple black-and-white squares pattern but a special one called ChArUco board, that is going to be described afterwards in detail.



To summarise, in this chapter the workstation setup, the hardware, and the software used for the image acquisition are first presented and they will be followed by a brief theoretical digression about the camera distortion model and the description of the ChArUco board. Then the code procedure to obtain the calibration parameters will be described by presenting the main functions used in the code.

2.2 Hardware and software

The workstation for this first part of the project consists of a platform, a vertical support, a ring light, and a camera: the support is placed perpendicularly to the platform, and the camera is mounted on the support at 90 cm from the platform so that the centre of the lens results above the centre of ring light (Figure 2.1). This guarantees the platform illumination without projecting on it the shadow of the camera and, with this configuration, the lamp does not interfere with the camera view.

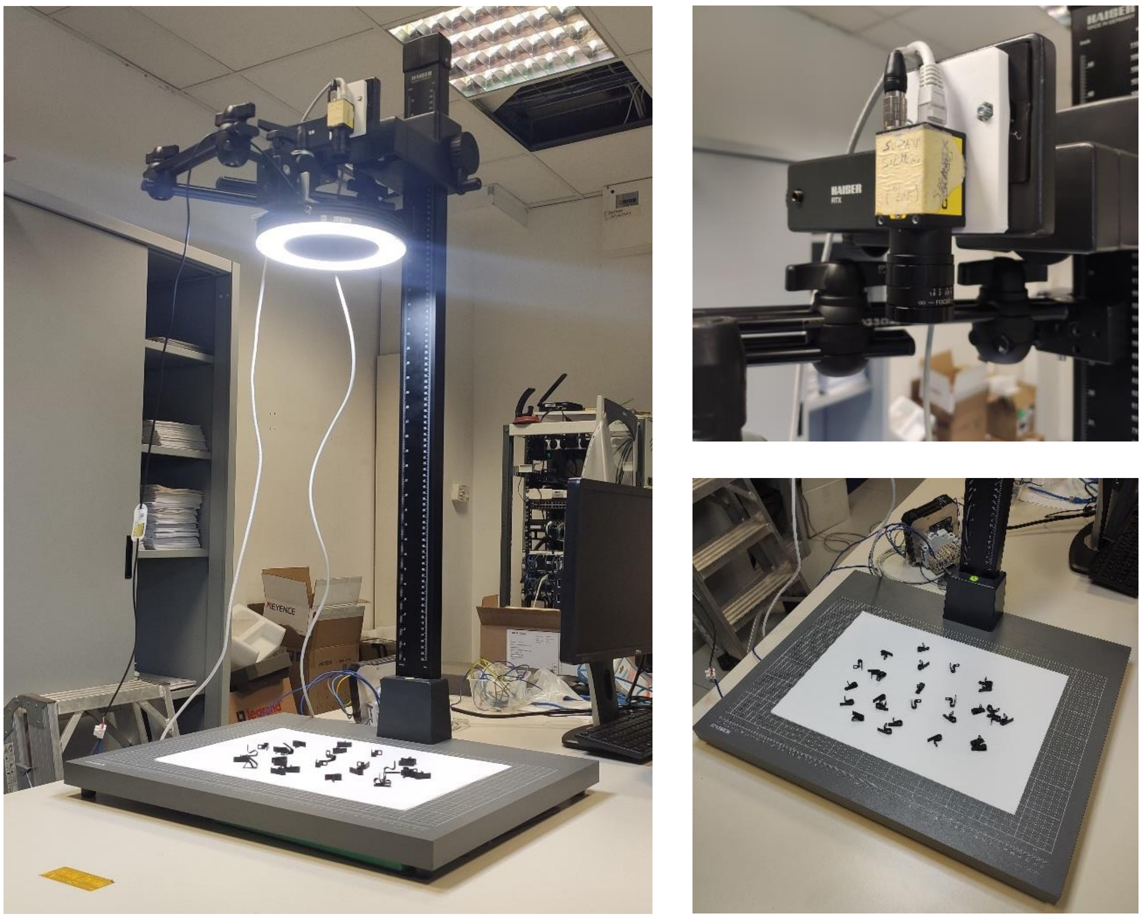


Figure 2.1: *workplace setup. The full equipment (left) includes the platform on whitch the pieces are randomly distributed (bottom right), a ring light and the camera (top right)*

Concerning hardware, the camera adopted is a streaming camera Basler acA2500-14gm, revised and commercialized by Cognex as CAM-CIC-5000-24-CG: it is characterized by GenICam protocol compatibility, and data interface GigE Vision, where GigE is an

interface standard introduced in 2006 for high-performance industrial cameras, developed applying the Gigabit Ethernet communication protocol and widely used around the world since it allows fast image transfer using low-cost standard cables, assuring robust performance over very long lengths. The lens is an Edmund Optics TechSpec-33304.
The three devices are shown in Figure 2.3.

Figure 2.2*: GigE logo [3]*

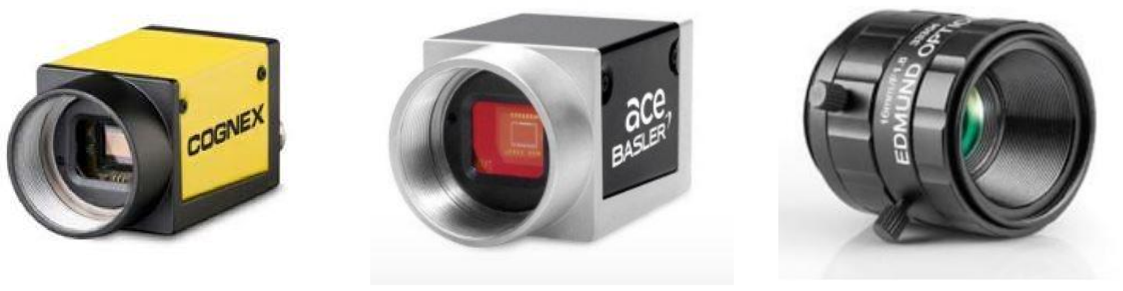


Figure 2.3*: camera and lens. From left to right: Cognex CAM-CIC-5000-24-CG [4], Basler acA2500-14gm [5], and Edmund Optics TechSpec-33304 [6]*

As for software, to write a python code able to communicate with the camera and acquire images, it has been implemented Harvester, a free-use Python library for image acquisition process in computer vision applications, which main features are:

- Image acquisition through GenTL Producers (libraries that have C interface and offer consumers a way to communicate with cameras over physical transport layer dependent technology hiding the detail from the consumer)

- Multiple loading of GenTL Producers in a single Python script.

- GenICam (Generic Interface for Cameras) feature node manipulation, where GenICam protocol provides generic programming interface for all kinds of devices, regardless of their interface technology (GigE, ethernet in our case, but also USB3 cameras) or what features they implement, as long as they are compliant to the GenICam standard.

Figure 2.4*: GenICam logo [7]*

This library has been fundamental to connect us to the camera, acquiring the images and changing the principal camera parameters such as exposure time and brightness (gain).

2.3 Distortions, intrinsics, extrinsics

Since we are dealing with just one camera, it’s possible to adopt the so-called pinhole camera model to ideally describe the mathematical relationship between the 3D real world coordinates of a point and its projection onto the 2D image plane.
However, since this model does not consider the presence of a real lens, the obtained image results to be distortion-free while, it’s well-known that in real-life applications, pinhole cameras introduce distortions to images in a significant way.

There are two major kinds of distortion, radial and tangential:

- Radial distortion: it makes straight lines appear curved and its effect is more significant as a point is more distant from the centre of the image (Figure 2.5). It can be mathematically described as:

$$x_{radial\_distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.1}$$

$$y_{radial\_distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.2}$$

$$r^2 = x^2 + y^2 \tag{2.3}$$

- Tangential distortion: it makes images appear nearer than expected because of the non-perfectly parallel alignment between the lens and the imaging plane. It can be mathematically described as:

$$x_{tangential\_distorted} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \tag{2.4}$$

$$y_{tangential\_distorted} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{2.5}$$

The parameters of this first set ($\mathrm{k_1},\ \mathrm{k_2},\ \mathrm{p_1},\ \mathrm{p_2},\ \mathrm{k_3}$) are known as distortion coefficients.

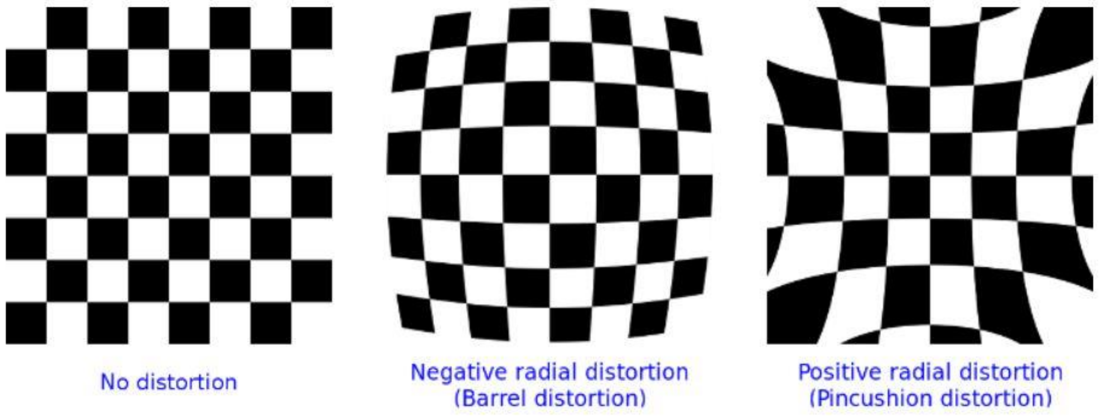

No distortion
Negative radial distortion (Barrel distortion)
Positive radial distortion (Pincushion distortion)

***Figure 2.5**: radial distortion examples [8]*

Anyway, two more sets of parameters are necessary for the calibration, and they are known as intrinsic and extrinsic parameters of the camera:

- Intrinsic parameters: they are specific to a camera, and they can be gathered into a 3X3 matrix called camera matrix, usually noted as A or K:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

  where (f_x, f_y, c_x, c_y) are the focal length and the optical centre of the camera. This matrix, unique for each specific camera and lens combination, is used to correct the image centre position and possible rotation due to focal length, mapping the coordinates of the 3D points in the camera system to the pixel coordinates in the 2D image system.

- Extrinsic parameters: they are gathered into a 4X4 roto-translation matrix, mapping the coordinates of a 3D real point into another coordinate system, for example the 3D camera system.

Concerning the camera pinhole model and the different coordinates systems that characterises it, a more thorough and complete analysis will be debated in the next chapter, while dealing with pose estimation.

2.4 ChArUco board

A ChArUco board is a chessboard-like pattern that combines a classic black-and-white squares chessboard with special markers called ArUco. They are synthetic square markers composed of a black squared background and an inner white binary matrix that determines the marker identifier, denoted as the marker id.

One of the most important characteristics of ArUco markers is their well-defined orientation: given a generic ArUco marker, its four corners are identified and listed in a specific clockwise order from 0 (top left corner in the original order) to 3.
This listing order does not change if the marker is rotated, implying that, if a reference frame is associated to a corner, it will remain fixed in that corner, no matter the orientation of the board.

ChArUco boards combine the benefits of both classic chessboard and ArUco markers (Figure 2.6): corners can be refined more accurately as for chessboards but their detection results to be faster because of ArUco markers properties. Furthermore, the property for which all the ArUco marker corners maintain their listing order no matter the rotation of the board is still valid, and, because of markers versatility, some occlusions or partial views don't compromise the detection of the board, conversely to what happens if a simple chessboard is adopted instead.

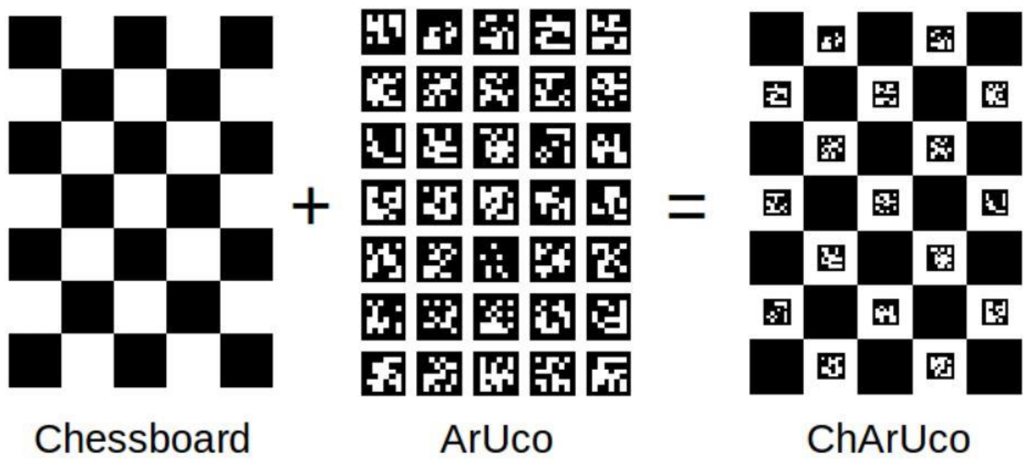


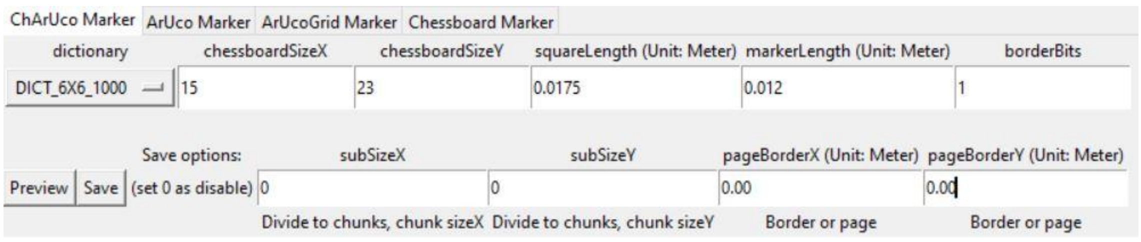
***Figure 2.6**: ChArUco board composition [9]*

In the program section, it will be described how to create a ChArUco board and how to define it in the code.

2.5 Code

The first step consists in acquiring images with the camera, with the functions provided by the Harvesters library that also consents to regulate exposure time, brightness (gain) and pixel format.

The second step consists in defining and printing the board.
To create it, it has been used the python codes MarkerPrinter.py and MarkerPrinterGUI.py provided by Josh Chien [10] that allows the user to select all the board geometrical parameters as dictionary, rows and columns number, square and marker length (in meters) and page border sizes (in meters), where the dictionary is a predefined set that indicates the number of bits and markers contained in the board. For example, DICT_6X6_1000 means that the dictionary is composed by 1000 markers, with size 6x6 bits (Figure 2.7).



dictionary	chessboardSizeX	chessboardSizeY	squareLength (Unit: Meter)	markerLength (Unit: Meter)	borderBits
DICT_6X6_1000	15	23	0.0175	0.012	1

Save options:	subSizeX	subSizeY	pageBorderX (Unit: Meter)	pageBorderY (Unit: Meter)
(set 0 as disable)	0	0	0.00	0.00
	Divide to chunks, chunk sizeX	Divide to chunks, chunk sizeY	Border or page	Border or page

Figure 2.7: MarkerPrinterGUI.py interface example [11]

To select the best combination of these parameters, different boards have been printed, considering that the total number of markers must not to be too low (lower precision in the pose estimation process) or too high (markers too small to be detected) and must be contained in a A3 paper sheet (297X420 mm).
From these attempts it has been noticed that the program correctly detects all the markers if the top and bottom left and right squares are black (even number of rows and columns) and, at the end, the board with the following parameter has been adopted (Figure 2.8):

- Dictionary: DICT_6X6_1000
- Rows: 15
- Columns: 23
- Square length [m]: 0.0175
- Marker length [m]: 0.012
- Border page x [m]: 0.01
- Border page y [m]: 0.01

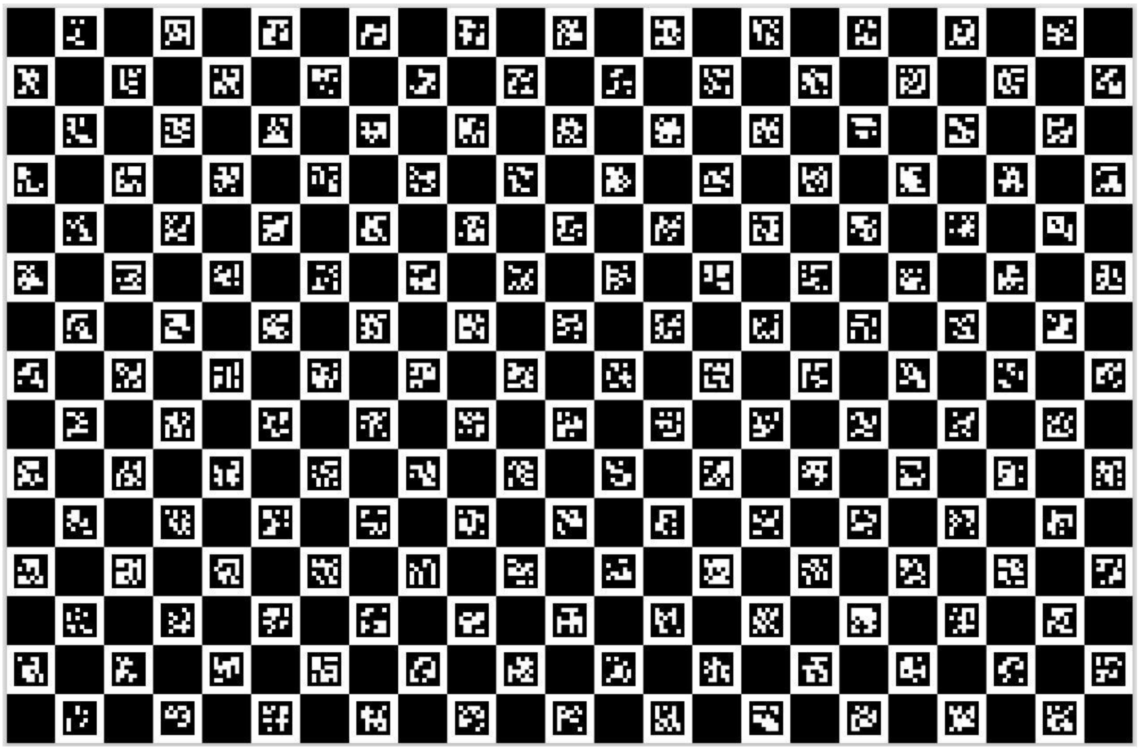


***Figure 2.8**: definitive ChArUco board*

Those parameters, except on the border page x and y, are used to describe the board in the code, by creating a dedicated class with the function cv.aruco.CharucoBoard():

***Function 2.1**: cv.aruco.CharucoBoard() [12]*

• Input:
  o **(ROWS, COLS)**: number of rows and columns of the board.
  o **SQUARE_LENGHT**: in millimetres.
  o **MARKER_LENGHT**: in millimetres.
  o **aruco_dict**: board dictionary.
 	
• Output:
  o **board**: board class.

Once the chessboard has been printed and the program section to acquire images is ready, it is finally possible to start the calibration procedure, where the program finds the chessboard corners, i.e., the points where two black squares touch each other, and, by

knowing the size of a square in mm, converts their 2D image coordinates into 3D real world coordinates. The first set of 2D points is known as image points set, while the second set is known as object points set.
In this case, since the board is planar, also the object points set result to be a 2D points set.

First, to achieve a good result, it is mandatory to take several pictures of the chessboard in different positions and with different orientations (at least 10), so, 20 different images of the board have been acquired for this phase (Figure 2.9).

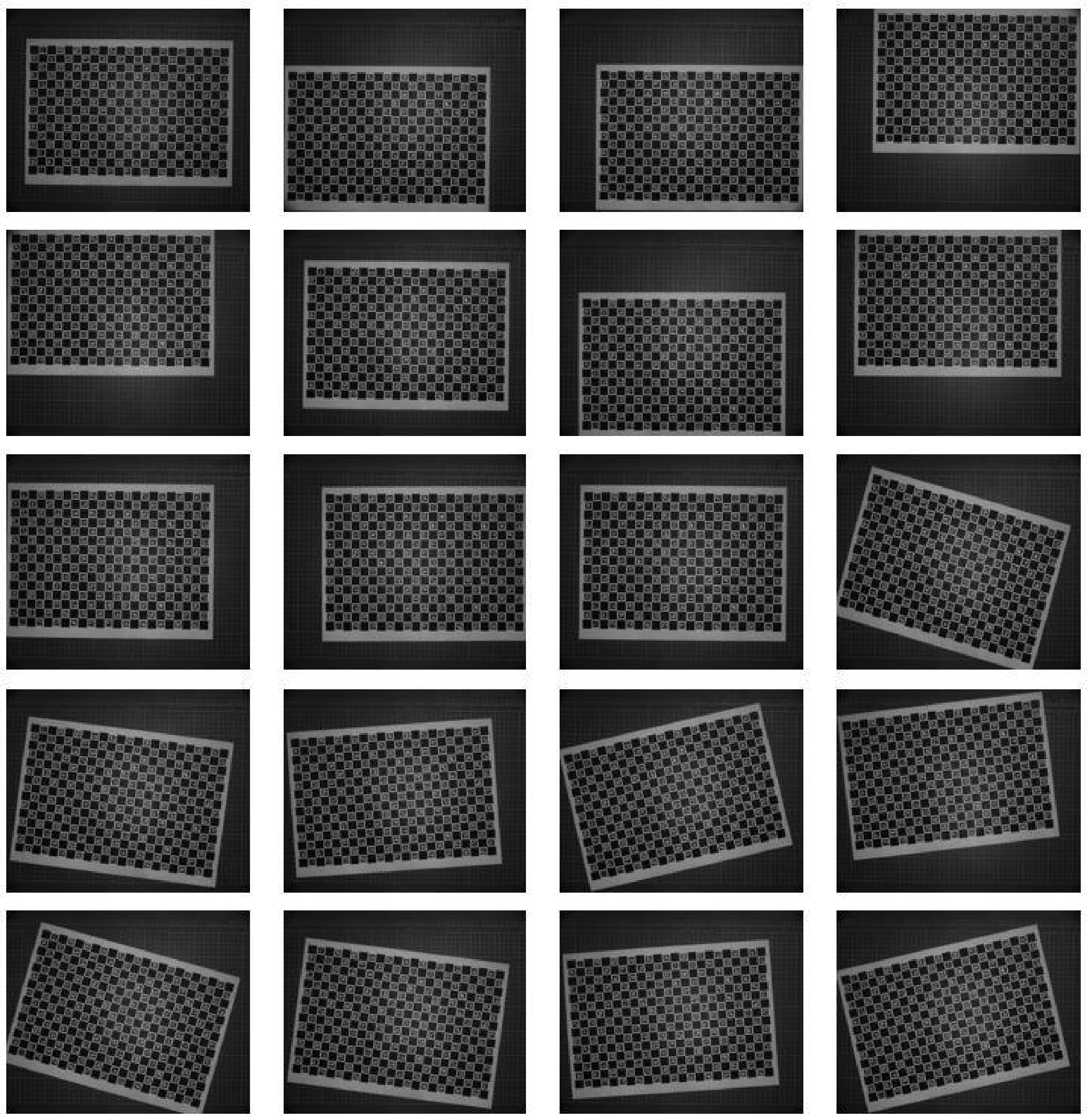


***Figure 2.9**: board calibration images*

For each image, a grey version is produced, then, the program detects the markers (shown in Figure 2.10) from the grey image with the function cv.aruco.detectMarkers():

***Function 2.2**: cv.aruco.detectMarkers() [13]*

- Input:
  - **image**: greyscale of image.
- Output:
  - **markerCorners**: list of the pixel coordinates of the corners of the detected markers, returned, for each marker, in clockwise order, starting with top left.
  - **markerIds**: list of detected marker ids.
  - **rejectedImgPoints**: list of possible invalid markers.

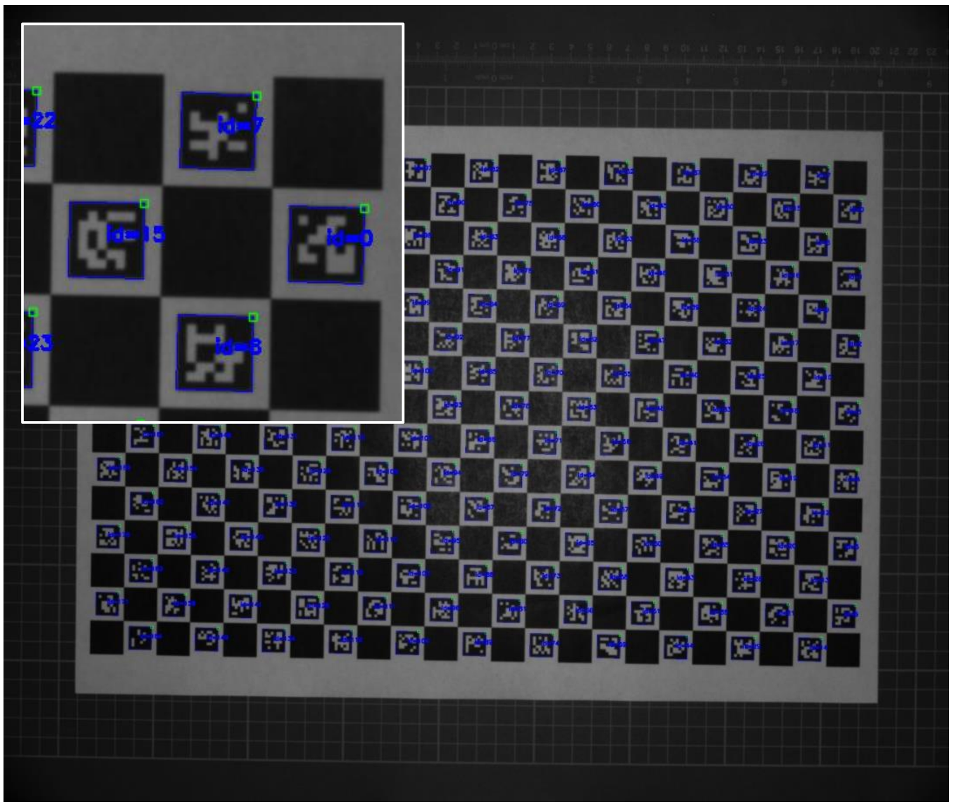


***Figure 2.10**: ArUco markers. In the detail (top left) it can be seen in a clearer way the detected markers (blue squares), their ordered ids (blue numbers), and the top left corner of each marker identified with a pink square*

Therefore, the marker corners, the corners ids, the grey image, and the board are used as inputs for the function cv.aruco.interpolateCornersCharuco() to obtain the interpolated ChArUco corners (308 corners, shown in Figure 2.11), by calculating the homography correlation between ChArUco plane and image projection.

***Function 2.3**: cv.aruco.interpolateCornersCharuco() [14]*

- Input:
  - **markerCorners**: from cv.aruco.detectMarkers().
  - **markerIds**: from cv.aruco.detectMarkers().
  - **image**: greyscale image.
  - **board**: as defined in cv.aruco.CharucoBoard.
- Output:
  - **resp**: number of detected squares.
  - **charucoCorners**: list of pixel coordinates of the interpolated chessboard corners.
  - **charucoIds**: list of the interpolated chessboard corners identifiers.

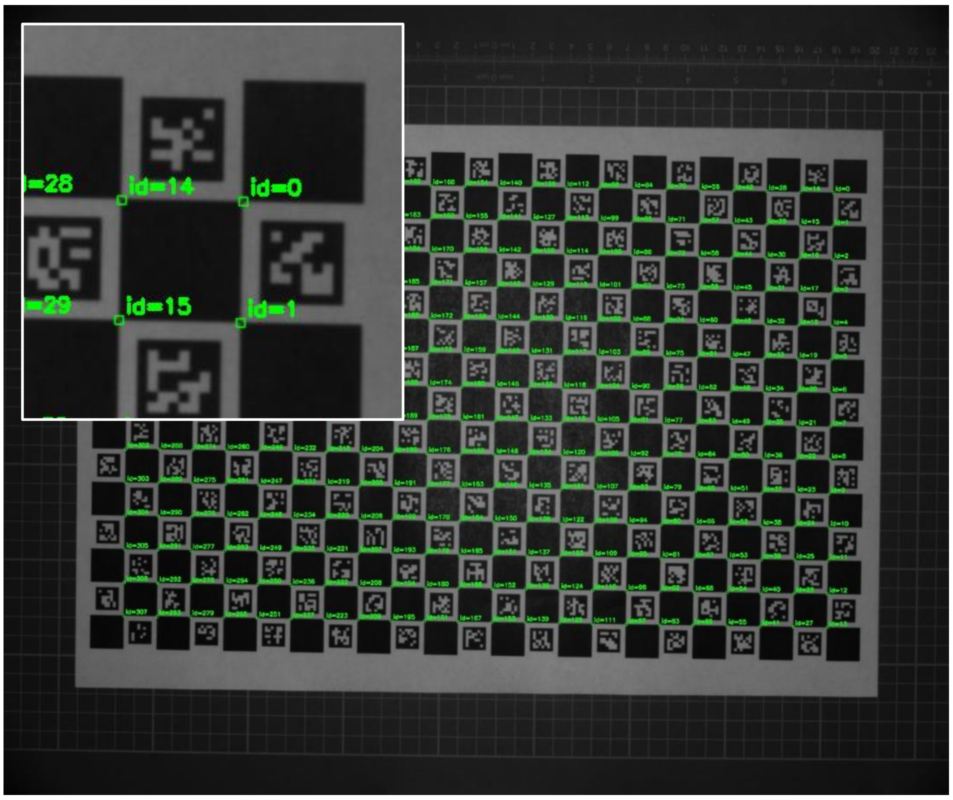


***Figure 2.11**: ChArUco corners. In the detail (top left) it can be seen in a clearer way the detected corners, identified by green squares and their ordered ids (green numbers)*

If the number of squares detected (resp) is greater than a threshold chosen by the user (20 in this case), the corners and their ids of all the images are collected in lists. Corner list, id list, board, and image size are finally used as inputs in the function cv.aruco.calibrateCameraCharuco() to get the camera matrix and the distortion coefficients:

Function 2.4: *cv.aruco.calibrateCameraCaruco() [15]*

- Input:
  - **charucoCorners**: list of all the interpolated corners.
  - **charucoIds**: list of all the interpolated corners identifiers.
  - **board**: as defined in cv.aruco.CharucoBoard.
  - **imageSize**: input image size.
- Output:
  - **cameraMatrix**: matrix K as previously defined.
  - **distCoeffs**: distortion coefficients $(k_1, k_2, p_1, p_2, k_3)$ as previously defined.

The result is the acquisition of both camera matrix and distortion coefficients set:

$$K = \begin{bmatrix} 63784.7042 & 0 & 1228.26881 \\ 0 & 63468.4528 & 1022.51891 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{distCoeff} = \begin{pmatrix} -15.0520757 \\ -0.945085157 \\ 0.0133808090 \\ 0.0444127560 \\ -0.000462258376 \end{pmatrix}$$

One last step before applying the corrective parameters to the image consists in obtaining a new camera matrix based on a free scale parameter alpha between 0 and 1 with the function cv.getOptimalNewCameraMatrix(), where 0 means that only sensible pixel from the original image are retrieved, while 1 means that all the original image pixels are kept.

***Function 2.5**: cv.getOptimalNewCameraMatrix() [16]*

- Input:
  - **cameraMatrix**: from cv.aruco.calibrateCameraCharuco().
  - **distCoeffs**: from cv.aruco.calibrateCameraCharuco().
  - **imageSize**: input image size.
  - **alpha**: set at 1.
  - **newImgSize**: input image size.
- Output:
  - **newCameraMatrix**: new matrix K.

The new camera matrix is:

$$\mathrm{K} = \begin{bmatrix} 63136.6493 & 0 & 1232.28102 \\ 0 & 62738.8758 & 1023.49849 \\ 0 & 0 & 1 \end{bmatrix}$$

Now, given the final camera matrix and the distortion coefficients, it is possible to see the result of this correction applied to one of the images, for example the first one, by using the function cv.undistort():

***Function 2.6**: cv.undistort() [17]*

- Input:
  - **src**: distorted image.
  - **cameraMatrix**: from cv.getOptimalNewCameraMatrix().
  - **distCoeffs**: as defined in cv.aruco.calibrateCameraCharuco().
  - **cameraMatrix**: from cv.getOptimalNewCameraMatrix().
- Output:
  - **dst**: corrected image.

The result of this last step can be finally seen in Figure 2.12, where the distorted and the undistorted images are compared: the fact that the two pictures are almost indistinguishable is a proof of the high camera-lens combination quality.

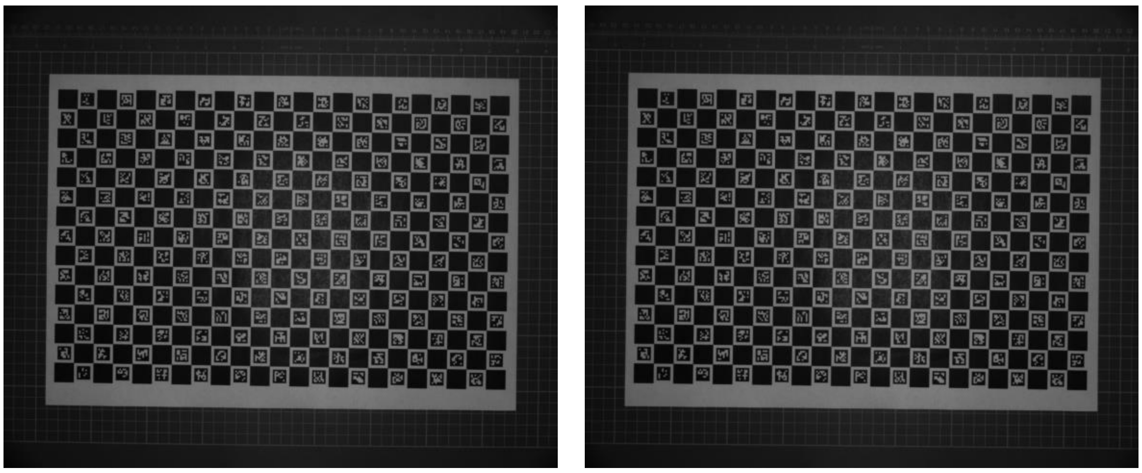
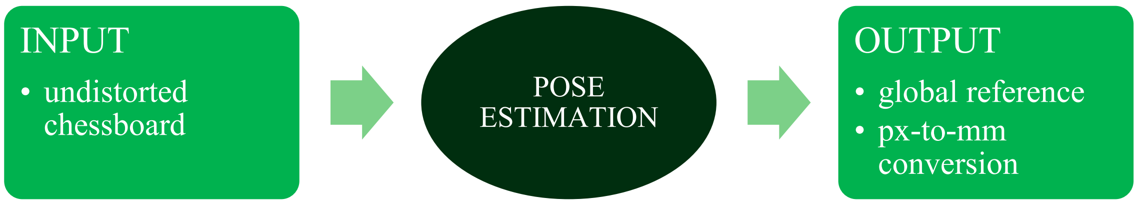


Figure 2.12*: undistorted image and distorted image comparison. Because of the high quality of the lens, the distorted image (left) is almost identical to the undistorted one (right)*

Chapter 3: pose estimation

3.1 Introduction

Once the intrinsic parameters of the camera are known (camera matrix and distortion coefficients), it is possible to use them to calculate the pose of an object in space, or rather its translation and rotation with respect to a coordinate system.
To perform this computation, it will be used the ChArUco board already adopted in the previous chapter: this operation result will lead to the definition of a global reference frame from which the pixel-to-millimetres conversion scale from the 2D image world to the 3D real world will be found.



One main assumption must be considered before starting: to simplify computations, the camera XY plane is assumed to be parallel to the world XY plane, or also the coordinate axis Z, perpendicular to the image plane, is considered coincident with the Z axis of the camera. This is true only if the camera is correctly placed above the platform, with very small rotations around the X and Y axes.
This means that, since the global coordinate system origin shall correspond to a corner of the board, there will be no translation along Z: the pixel-to-millimetres conversion will basically be a 2D problem.

It will be shown that one of the functions implemented also returns the camera rotation angles around X and Y: to obtain the desired setup configuration the user must then manually adjust the camera pose to set those angles as close as possible to zero.

In this chapter will describe in detail the pinhole camera model, already mentioned in the previous part, followed by the correlation between world system, camera system and image system. Then, the pose estimation procedure and its main functions will be discussed through the code.

3.2 Camera model

To better understand the pose estimation process, it is mandatory to introduce more in detail the pinhole camera model (represented in Figure 3.1), where the view of a scene is obtained by projecting 3D points into the image 2D world using a perspective transformation that involves three different cartesian reference systems, written in homogeneous form:

- $\mathrm{P_w} = (\mathrm{X_w} \quad \mathrm{Y_w} \quad \mathrm{Z_w} \quad 1)^\mathrm{T}$: world (global) reference frame, representing the coordinates of a point in the real 3D world in millimetres. It is useful to indicate it also in the standard form as $\mathrm{p_w} = (\mathrm{X_w} \quad \mathrm{Y_w} \quad \mathrm{Z_w})^\mathrm{T}$.

- $\mathrm{P_c} = (\mathrm{X_c} \quad \mathrm{Y_c} \quad \mathrm{Z_c} \quad 1)^\mathrm{T}$: camera reference frame, representing the coordinates of a point with respect to the lens of the camera in millimetres, where its standard form is $\mathrm{p_c} = (\mathrm{X_c} \quad \mathrm{Y_c} \quad \mathrm{Z_c})^\mathrm{T}$.

- $\mathrm{p} = (\mathrm{u} \quad \mathrm{v} \quad 1)^\mathrm{T}$: image reference frame, representing the coordinates on a point in the 2D image world in pixels, where the origin conventionally corresponds to the top-left corner of the image.

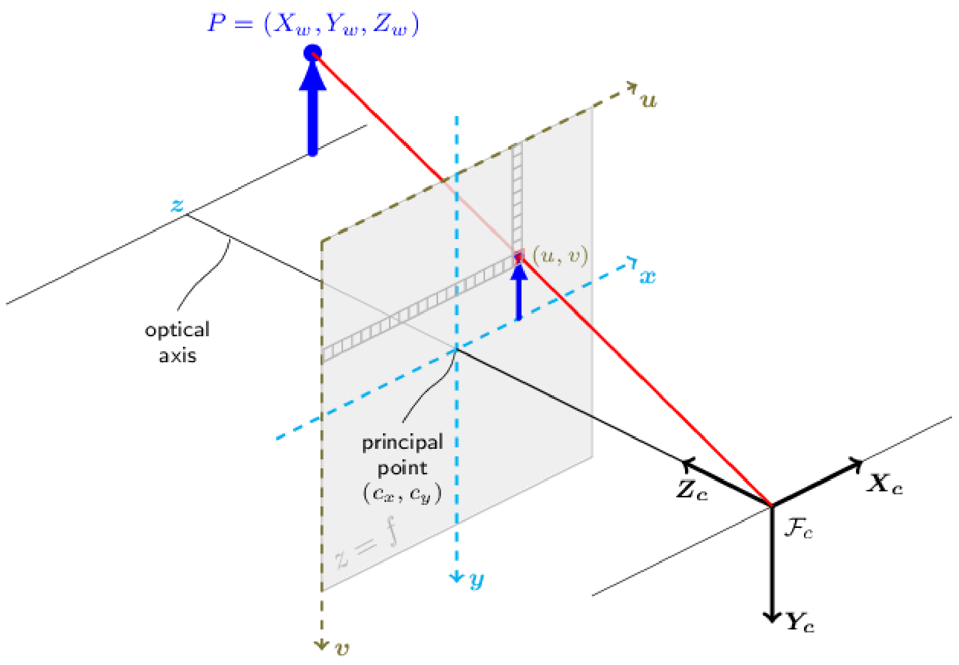


Figure 3.1: *pinhole camera model [18]*

Starting from a point with known global coordinates, its coordinates in the camera frame are obtained as:

$$P_c = R_t P_w = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} P_w \tag{3.1}$$

where R_t is the roto-translation matrix between the two frames, R is the rotation matrix, and t the translation vector. The form [R|t] will be used in the Equation 3.5:

$$R_t = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

$$\downarrow$$

$$[R|t] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \tag{3.3}$$

The passage from camera frame to image frame is then performed by multiplying the camera matrix K by the camera coordinates in standard form:

$$sp = Kp_c = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} p_c \tag{3.4}$$

where s is an arbitrary projecting scale factor.
Finally, the complete transformation from global frame to image frame can be written as:

$$sp = K[R|t]P_w = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} P_w \tag{3.5}$$

Figure 3.2 shows the two main frames useful for the conversion, where the origin of p coincides with the top-left corner of the image, while the origin of p_w is placed on the image plane, Z_w is perpendicular to it and the other two axes are oriented according to the board pattern.

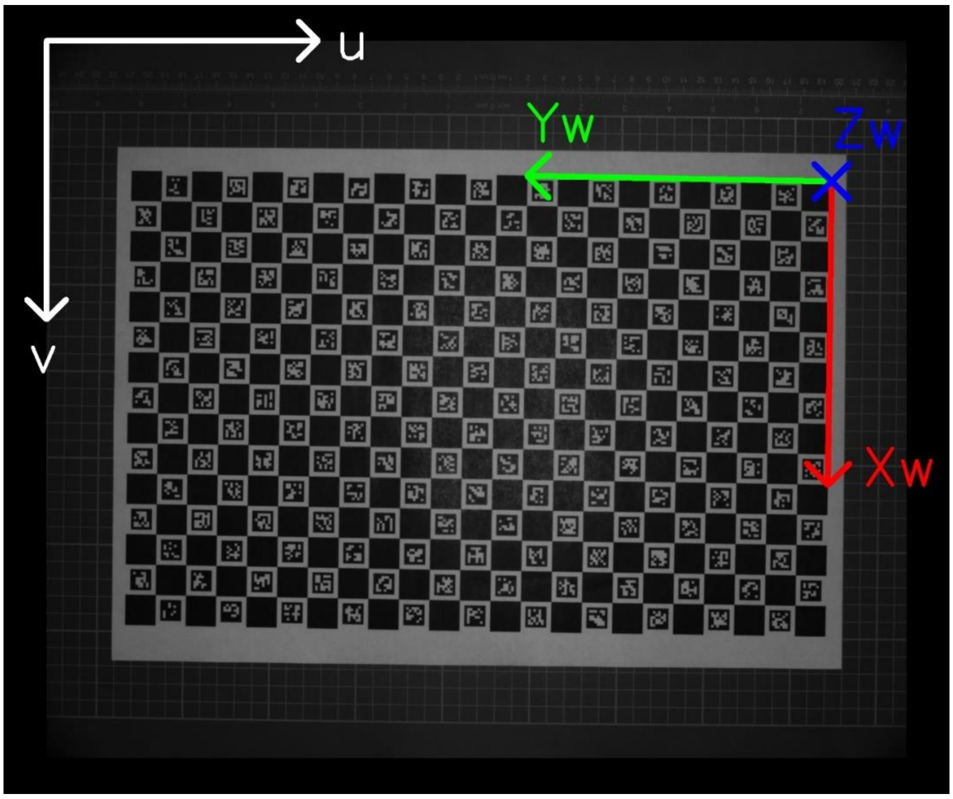


Figure 3.2*: reference frames representation. The image frame (u v) has the origin coincident with the top left corner of the image, while the global frame (X_w Y_w Z_w) has the origin coincident with the top right edge of the board and the Z axis entering the page.*

This is the theory behind the transformation from 3D real points to 2D image points. However, the goal now is to obtain the inverse relationship since, in the end, the program shall be able to transform a point in the image plane into a set of real coordinates for the robot.

First, the Equation 3.5 can be rewritten as:

$$sp = K(Rp_w + t) \tag{3.6}$$

$$\downarrow$$

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 0 \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \right) \tag{3.7}$$

where $\mathrm{Z_w} = 0$.
Then, through the following passages, it is possible to find $\mathrm{X_w}$ and $\mathrm{Y_w}$ by inverting the matrices K and R:

$$sp = K(Rp_w + t) \rightarrow sp = KRp_w + Kt \rightarrow sp - Kt = KRp_w \tag{3.8}$$

$$\downarrow$$

$$p_w = (KR)^{-1}(sp - Kt) \tag{3.9}$$

The only unknown term on the right side is the projecting scale factor s. Anyway, since $\mathrm{Z_w} = 0$, it can be easily obtained by looking at the last element of each matrix product:

$$s[(KR)^{-1}p]_3 - [(KR)^{-1}Kt]_3 = 0 \tag{3.10}$$

$$\downarrow$$

$$s = \frac{[(KR)^{-1}Kt]_3}{[(KR)^{-1}p]_3} \tag{3.11}$$

where $[V]_3$ is the third (last in this case) element of the generic vector V.

3.3 Code

The pose estimation code is the continuation of the calibration part: again, one of the board images, for example the first one, is chosen as reference: the distortion correction is applied and, as already done in the previous chapter, all the corners ad their Ids are detected with the function cv.aruco.interpolateCornersCharuco().
Then, the function cv.aruco.estimatePoseCharucoBoard() is used to find the rotation vector and the translation vector of the board with respect to the origin of the image reference:

***Function 3.1**: cv.aruco.estimatePoseChacrucoBoard() [19]*

- Input:
  - **charucoCorners**: from cv.aruco.interpolateCornersCharuco().
  - **charucoIds**: from cv.aruco.interpolateCornersCharuco().
  - **board**: as defined in cv.aruco.CharucoBoard.
  - **cameraMatrix**: from cv.getOptimalNewCameraMatrix().
  - **distCoeffs**: from cv.aruco.calibrateCameraCharuco().

- Output:
  - **retval**: Boolean value, true if the pose estimation is valid, false otherwise.
  - **rvec**: rotation vector of the board in the Rodrigues notation.
  - **tvec**: translation vector of the board.

If retval is true, rvec and tvec are used in the function cv.drawframeAxes() [20] to draw the reference frame on the board: in Figure 3.3 it is possible to see how the origin is located on the top right corner of the board edge, the X axis is drawn in red, the Y axis is green and the Z axis is blue (entering the page).

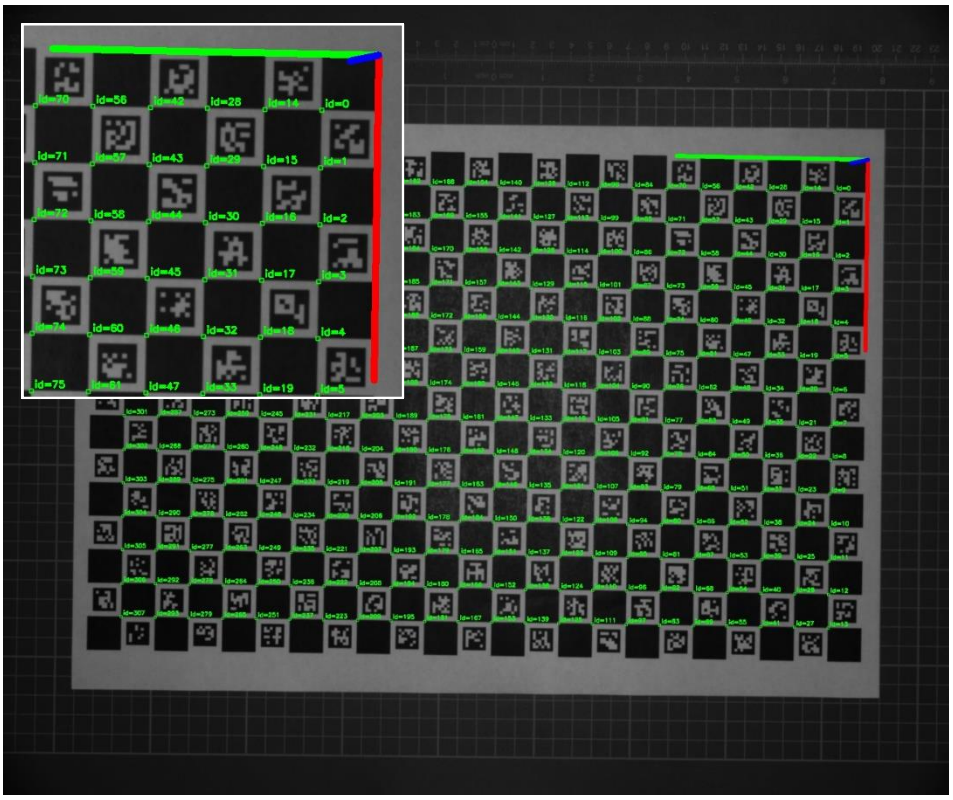


Figure 3.3: global reference frame

To have a good estimation of the intrinsic parameters correctness, the reprojection error is computed as the absolute norm between the coordinates of the interpolated chessboard corners charucoCorners, previously obtained from cv.aruco.interpolateCornersCharuco(), and the image points.
The image points, 2D reprojection of 3D points, are obtained from the object points (3D points) with the function cv.projectPoints() [21], in turn obtained by the function cv.aruco.getBoardObjectAndImagePoints():

***Function 3.2**: cv.aruco.getBoardObjectAndImagePoints() [22]*

- Input:
  - **board**: as defined in cv.aruco.CharucoBoard.
  - **charucoCorners**: from cv.aruco.interpolateCornersCharuco().
  - **charucoIds**: from cv.aruco.interpolateCornersCharuco().

- Output:
  - **objPoints**: coordinates in millimetres of board marker points.

The result of the norm is then divided by the number of image points to get the arithmetical average and, in the end, the reprojection error results to be equal to 0.06 px: by converting this error in millimetres, once obtained the conversion scale values, it will be shown that this error is quite small.

The next step consists into finding the rotation matrix R and the Euler angles, that are obtained first by applying the function cv.Rodrigues() [23] to rvec, which performs the Rodrigues inverse transformation, then, by applying to R the MATLAB function rotm2eul() [24] which returns the three rotation angles in the order Z-X-Y:

$$\mathrm{R} = \begin{bmatrix} -0.01522158 & -0.99732713 & -0.07146261 \\ 0.99983859 & -0.0158642 & 0.00843344 \\ -0.0095446 & -0.0713227 & 0.99740763 \end{bmatrix}$$

$$\downarrow$$

$$\theta_{\mathrm{ZXY}} = \begin{pmatrix} 1.5860 \\ 0.0095 \\ -0.0714 \end{pmatrix} [\mathrm{rad}] = \begin{pmatrix} 90.8722 \\ 0.5469 \\ -4.0901 \end{pmatrix} [^\circ]$$

Since the board is planar, possible non-zero angles around X and Y are consequences of an imprecise positioning of the camera, that results to be non-parallel to the board. As already mentioned at the beginning of the chapter, the user should manually correct the camera positioning and repeat the procedure until the resulting X and Y angles values are very close to zero, but this operation is not simple if executed without specific equipment. For this reason, few degrees rotation angles around X and Y are considered acceptable.
Now that K, R, and t are known, the last missing term s is obtained by applying the Equation 3.11. Then, it's finally possible to apply the Equation 3.10 to obtain the pixels-to-millimetres conversion for any desired point of the image.

To have an idea of the pixels-to-millimetres conversion scale, a simple geometrical procedure is proposed: because of the function cv.projectPoints(), the origin O and the extreme top-left (TL) and down-right (DR) corners pixel coordinates are well known so, it is possible to calculate the distance between these points and the origin:

$$\overline{DR,O} = \sqrt{(u_O - u_{DR})^2 + (v_O - v_{DR})^2} \tag{3.12}$$

$$\overline{TL,O} = \sqrt{(u_O - u_{TL})^2 + (v_O - v_{TL})^2} \tag{3.13}$$

Then, the real-world width distance (WD) and length distance (LD) correspond to:

$$WD = rows\ number * square\ length \tag{3.14}$$

$$LD = columns\ number * square\ length \tag{3.15}$$

The conversion scale values are then:

$$s_x = \frac{WD}{\overline{\overline{DR, O}}} \tag{3.16}$$

$$s_y = \frac{WL}{\overline{\overline{TL, O}}} \tag{3.17}$$

And the result is: $s_x = 0.20099$ [mm/px] and $s_y = 0.20035$ [mm/px].
The fact that the two scale values are so similar, is another proof that the camera rotation angles around X and Y are quite small, and so the board squares shapes are not affected in a significant way by perspective.

Given the conversion scale values it is possible to estimate the maximum value of the reprojection error in millimetres by multiplying it for the maximum between s_x and s_y, and, as previously mentioned, this error results to be quite small:

$$rep\_error_{mm} = rep\_error_{px} * max(s_x, s_y) = 0.06 * 0.20099 = 0.0120594\ [mm]$$

Finally, to estimate the error between the transformation results and the actual measurements, for each ChArUco corner of the board it has been computed the absolute value of the difference between the X, the Y, the corner-to-origin (diagonal) distance and their values directly measured on the printed board.
In Table 3.1 the maximum, the minimum and the average errors in millimetres for each coordinate are collected, while, Figure 3.4, 3.5, and 3.6, show the MATLAB plots of the errors for each corner, to give an idea of the error distribution all over the board.
From Table 3.1, the maximum error affecting the corners is less than 0.77 mm on the diagonal, that is a quite good result.

error [mm]	X	Y	DIAGONAL
maximum	0.5103	0.7286	0.7698
minimum	0.0010	0.0010	0.0004
average	0.1410	0.2595	0.2366

Table 3.1*: pose estimation errors*

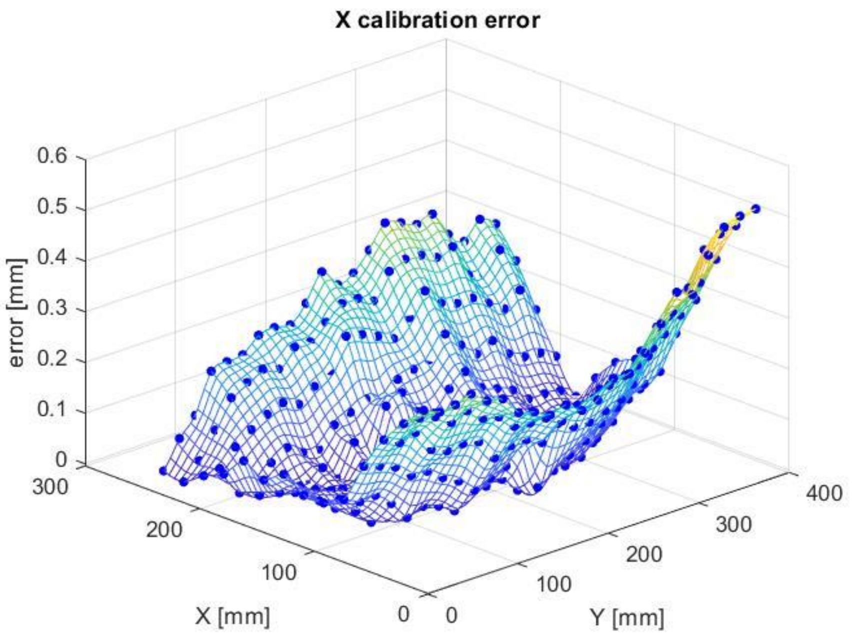


Figure 3.4*: X calibration error*

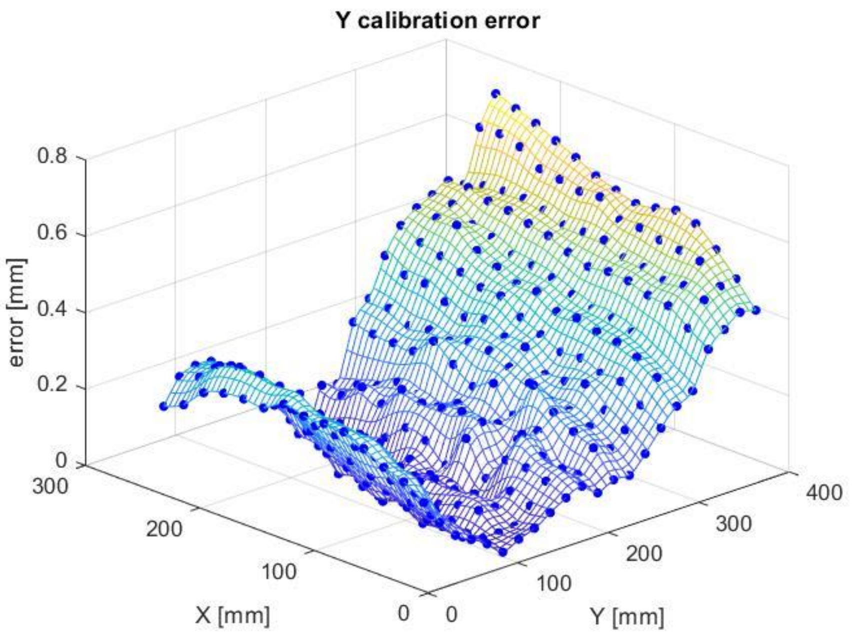


Figure 3.5: *Y calibration error*

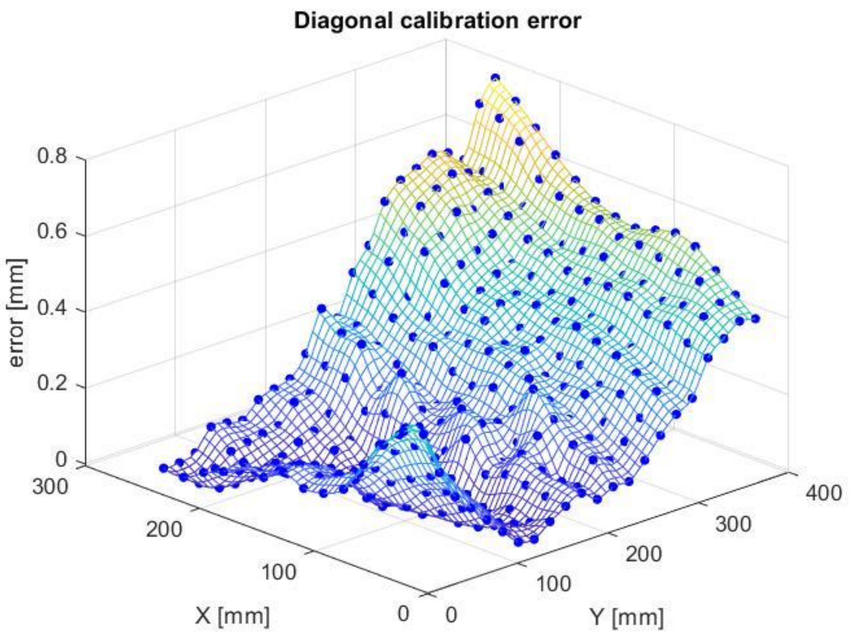


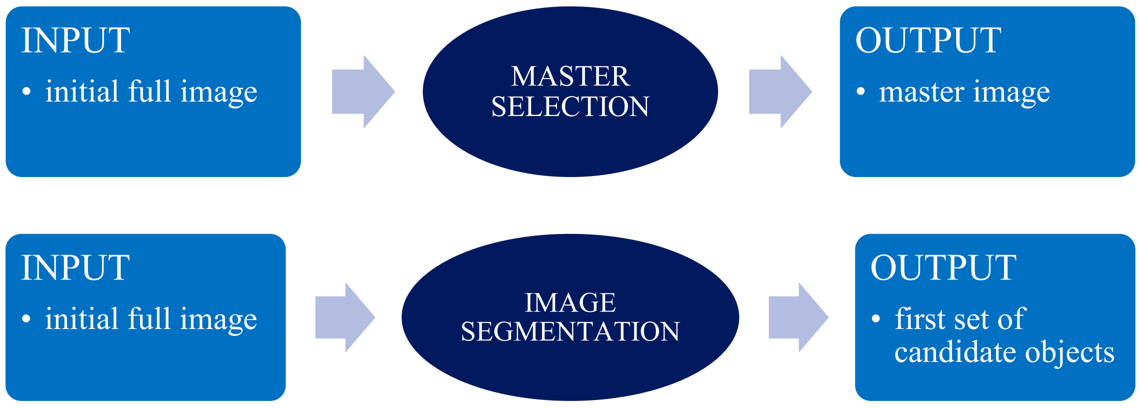
Figure 3.4: *diagonal calibration error*

Chapter 4: segmentation into object candidates

4.1 Introduction

In the previous two chapters, the preparatory steps for the development of the main part of this thesis project have been examined, that is a quite standard procedure for every computer vision work: in fact, the program is now ready to acquire distortion-free images of the platform and convert each pixel coordinates into real world millimetres coordinates, with respect to the global frame.

Being this the first chapter concerning the main program, it is discussed how, given a bunch of pieces randomly posed on the platform, the reference piece (the master) is chosen and how a first, preliminary set of candidate objects is obtained with a procedure that only involves the shape of the pieces, by comparing each piece with the master.
This procedure is based on a very largely used technique called image segmentation, which in turn is based on another fundamental image processing method called thresholding.



Differently from the previous two chapters, from this one on, it has been decided to explain step by step the whole project following the code implementation order. This means that, as a matter of clarity, theoretical explanations will not be discussed at the beginning of the chapter but gradually, when required.

4.2 Grip constraints definition and master selection

The very first step of this program consists, obviously, in the acquisition of the full image of the platform, that is automatically corrected by applying the results of the camera calibration process. Figure 4.1 shows the first image taken for the final tests section, that will be used as example from now on for all the code explanations.
The discussion will refer to this particular kind of R-shaped black pieces with the tag 340.

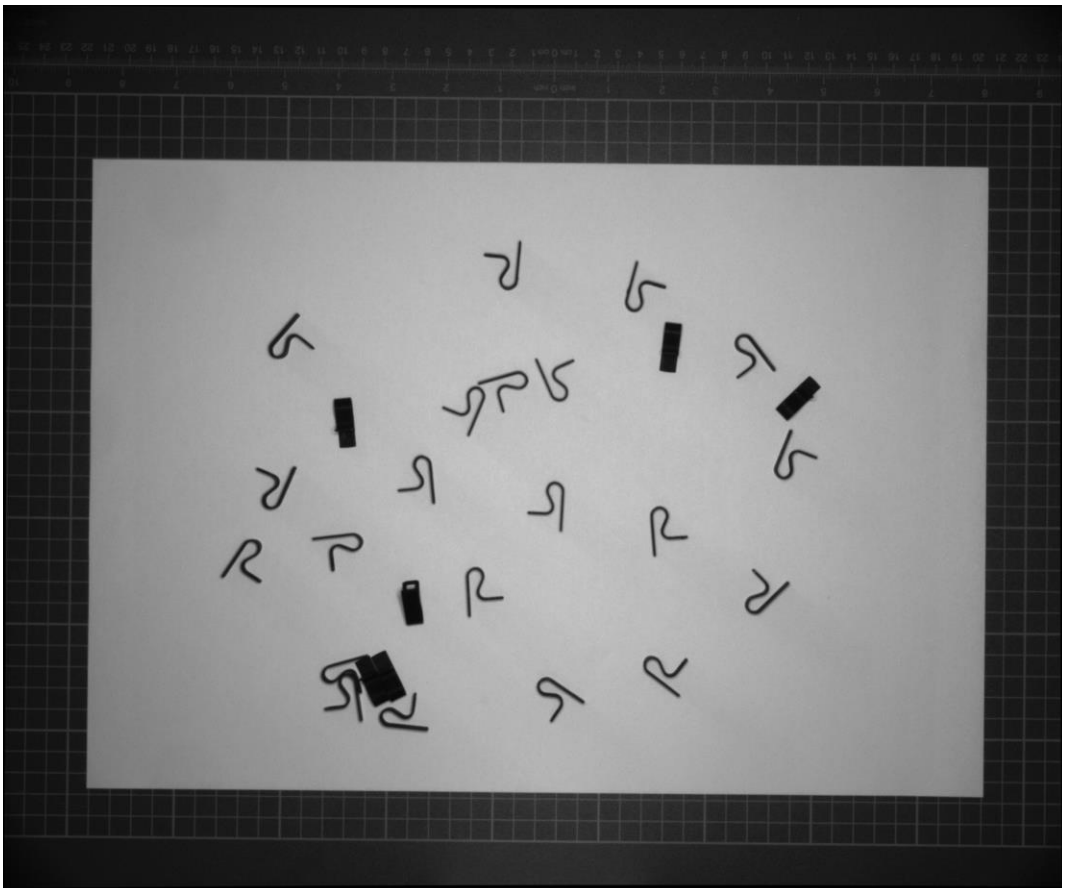


***Figure 4.1**: 340 full image*

Once the undistorted image has been acquired it is mandatory, before proceeding, to set some coordinates constraints on the image to exclude the areas close to the borders of the platform, to avoid a possible collision of the robot during the grip operation. These constraints are imposed by the user and are represented by four lines printed on the image as well as their coordinates (Figure 4.2). When the grip point of each piece will be defined, if one of them results to be located outside these constraints, the corresponding piece will not be considered for the grip.

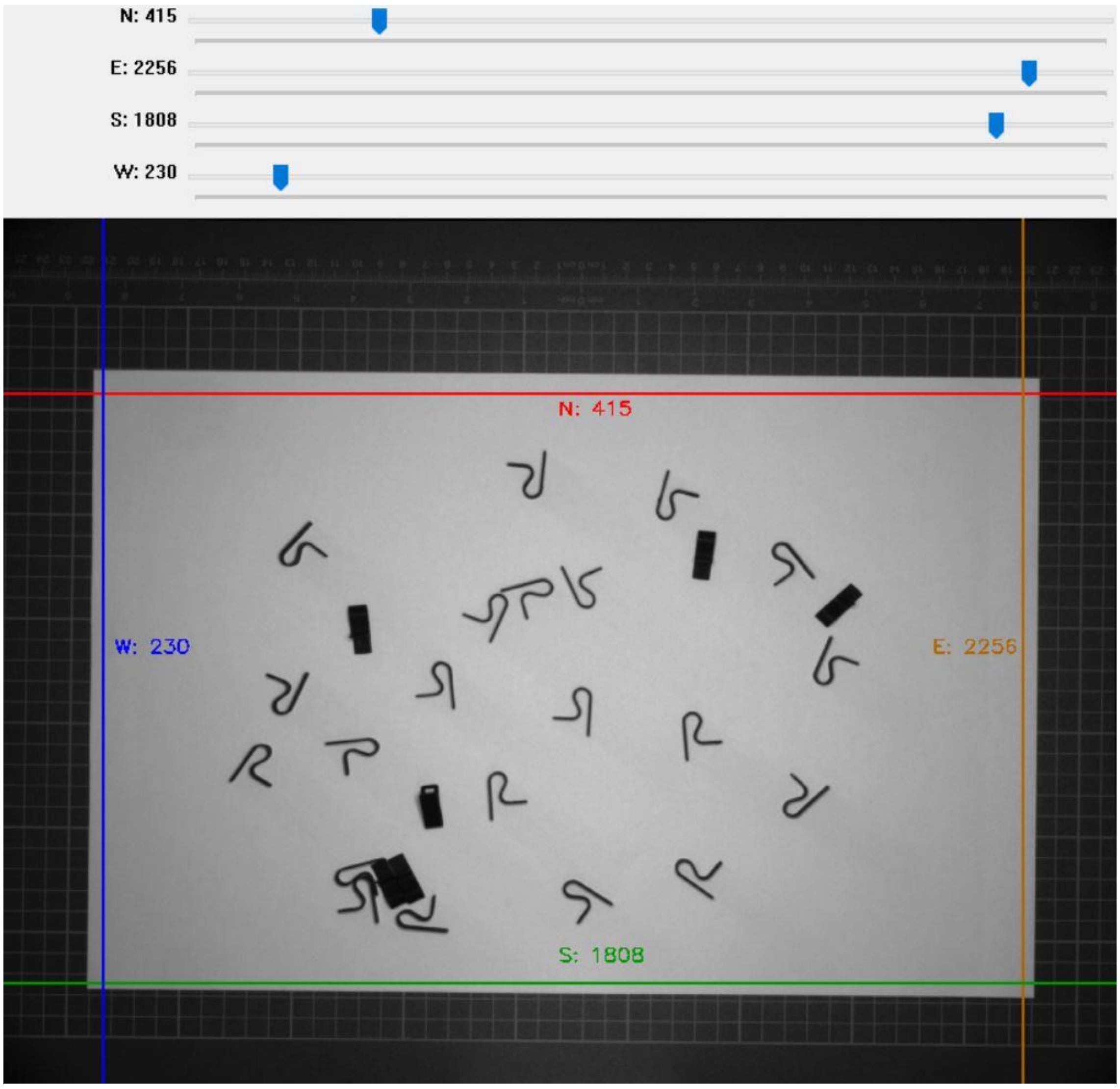


***Figure 4.2**: grip contraints. N stands for North (red line), E for East (orange line), S for South (green line), and W for West (blue line)*

Now, the user must choose the master, that is the reference piece: this is a crucial choice, since the master will be the reference not only for the pieces considered for this project discussion, but it will be saved and re-used as reference every time a new batch of pieces of the same kind is processed.
For this reason, the master must be a well-posed and well-illuminated piece, without other pieces or any sort of obstacles in its neighbourhood. That's why it is usually placed ad hoc on the platform, exclusively for this first operation.
Also, the master image (Figure 4.3) must be cropped with proper sizes: large enough to consent all the operations that are going to be analysed in detail in the next chapter, but not too large to cause ambiguity between close pieces during the feature matching phase.

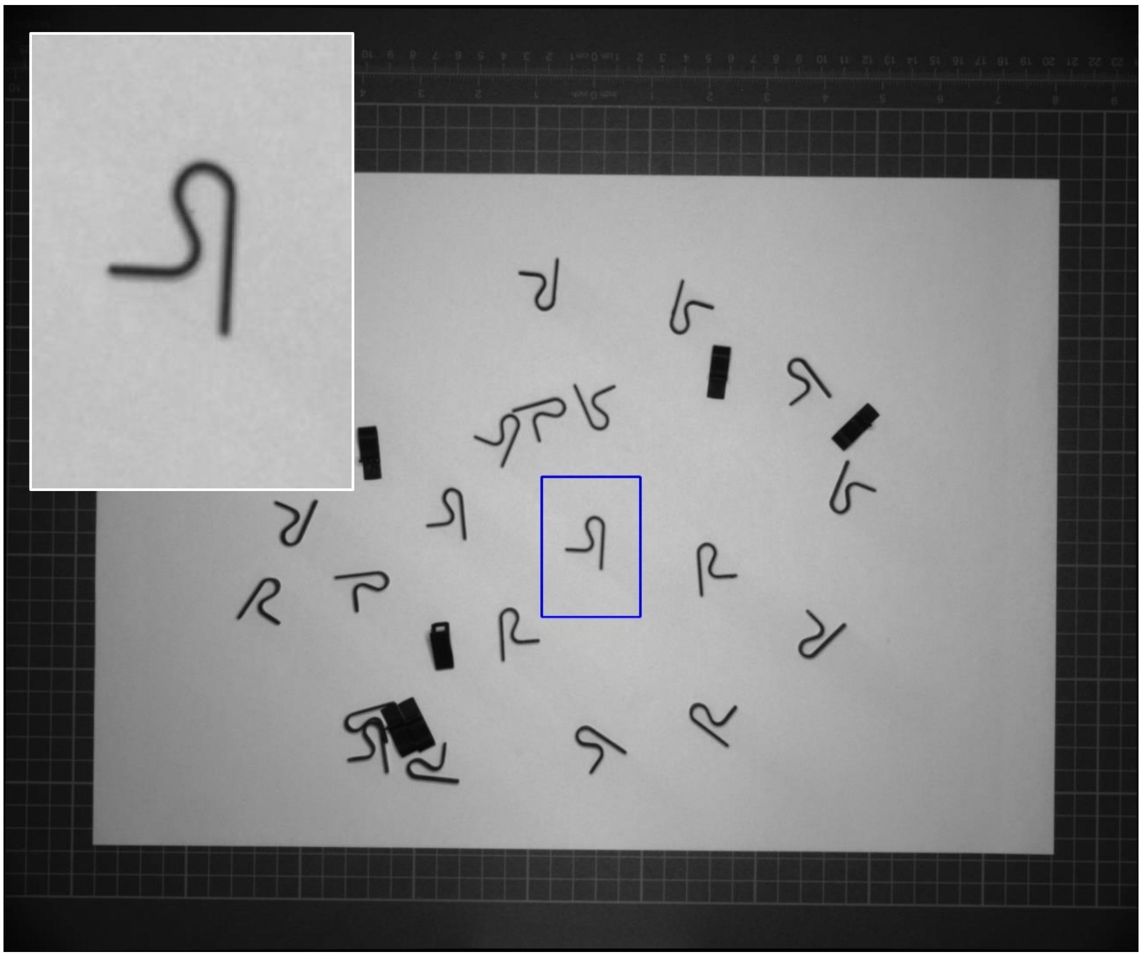


Figure 4.3*: choice of the reference piece. The detail (up left) shows the master as a result of the cropping action (blue rectangle) on the full image*

Since a lot of operations in the next phases are based on the sizes of the master image, the program executes a further step before proceeding over: to be sure that master image height and length are divisible by two, for a matter of precision, the program automatically corrects the cropping sizes manually imposed by the user, by adding a column or a row of pixels, if one of the two sizes results to be an odd number.

4.3 Segmentation

Segmentation is an image manipulation technique commonly used in computer vision applications since it consists in easily distinguishing an object from its background by detecting its corresponding silhouette, also called blob. The blob resulting from this operation is, in this case, a black spot shaped as the piece on a white background, as shown in Figure 4.4.

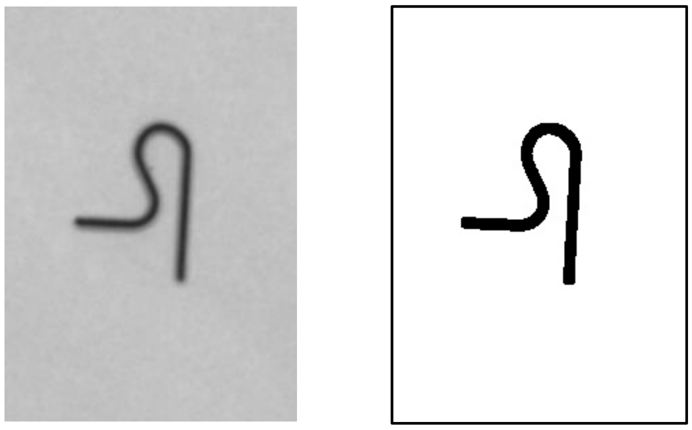


***Figure 4.4**: master piece and its corresponding blob*

For this project, the segmentation procedure can be schematized as a sequence of three operations, where the first, thresholding, is the most important one:

1) Thresholding of the master and of the full image to isolate the pieces from the background and obtain the blobs.

2) Noise removal by closing operation.

3) Blob grouping by erosion operation. This is necessary only if a piece is characterized by multiple blobs because of the illumination conditions and it is not possible to apply adaptive thresholding.

However, to better understand what thresholding is, and consequently how segmentation works, it is better to introduce some digital image basics.

4.4 Digital image basics

A digital image is nothing but a matrix of pixels, where each pixel is associated to a vector of values (channels) called colour space and where each channel assumes a numerical value (intensity). The combination of colour space kind, number of channels and intensity values determines the colour of the pixel.

Some of the most used colour spaces (Figure 4.5) are:

- BGR (blue, green, red): the colour space adopted by OpenCV, where the intensity of each channel goes from 0 to 255. To give some examples, the pixel characterized by the combination $[0, 0, 255]$ is a red pixel, while the pixel $[0, 255, 255]$ is a yellow pixel. Remarkable cases are black pixels $[0, 0, 0]$ and white pixels $[255, 255, 255]$.

- RGB (red, green, blue): the colour space adopted by Matplotlib, a very useful Python library, ideal to represent multiple images and check pixels coordinates. This colour space is the inverse of RBG, so the pixel $[0, 0, 255]$ will be blue. Black and white pixels are unchanged.

- GREYSCALE (black/white): black-and-white colour space, characterized by one single channel that can assume intensity values from 0 (black) to 255 (white). It is frequently used for a lot of image manipulation processes as thresholding.

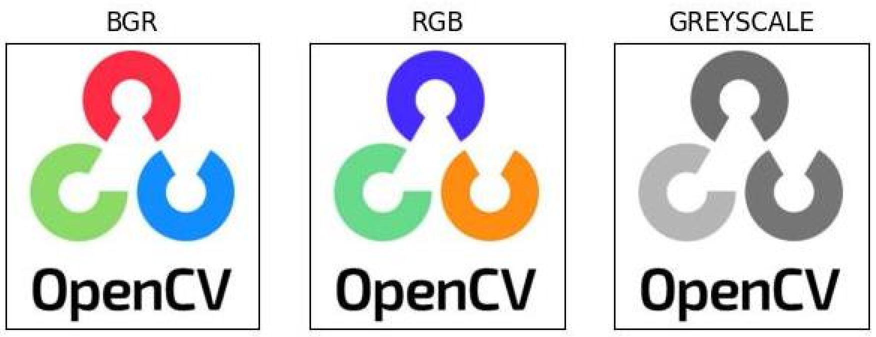


Figure 4.5: *OpenCV logo [25] printed with different colour spaces*

Since it can be useful to swap from a colour space to another, especially from colour to grey, OpenCV provides very simple methods to easily perform this change.

4.5 Thresholding

In digital image manipulation, thresholding is the simplest segmentation process. Given a greyscale image, it consists in replacing each pixel of the image with a white pixel if its intensity value $I_{i,j}$ is greater than a threshold T, otherwise it is replaced with a black pixel: this process is called binary thresholding, while the opposite is called inverted binary thresholding.

In formulas:

$$I_{i,j}^{bin} = \begin{cases} 255 \; if \; I_{i,j} > T \\ 0 \; otherwise \end{cases} \tag{4.1}$$

$$I_{i,j}^{inv} = \begin{cases} 0 \; if \; I_{i,j} > T \\ 255 \; otherwise \end{cases} \tag{4.2}$$

In general, thresholding methods can be subdivided in two major groups:

- Global thresholding: the threshold is applied to every pixel of the image, useful for homogeneous illumination conditions. Binary, inverted binary and other kinds as truncated, threshold-to-zero, and inverted threshold-to zero belong to this group (Figure 4.6).

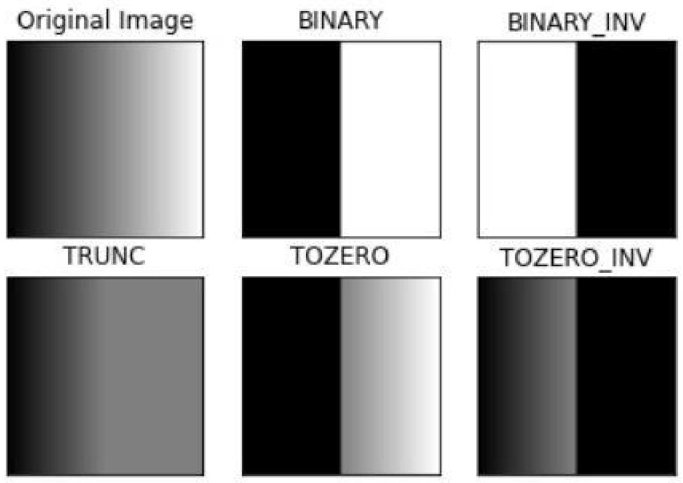


Figure 4.6*: global thersholding [26]*

- Adaptive (local) thresholding: an algorithm determines the threshold to be applied to each pixel basing the computations on the mean of a small neighbourhood region around the pixel itself or on the Gaussian-weighted sum of the neighbourhood values. It is useful when the image is characterized by different lighting conditions (Figure 4.7).

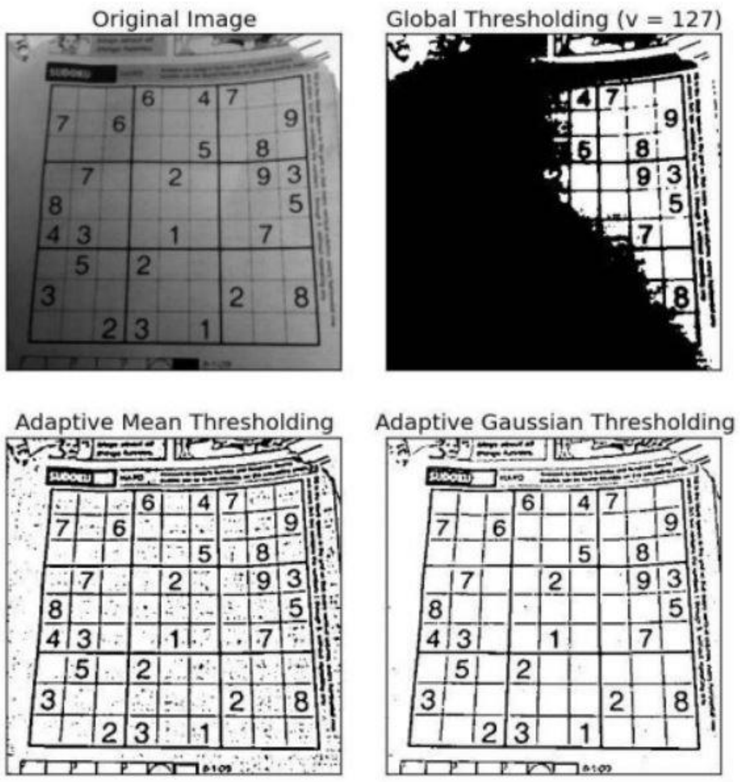

Original Image
Global Thresholding (v = 127)
Adaptive Mean Thresholding
Adaptive Gaussian Thresholding

Figure 4.*7: global and adaptive thresholding comparison [27]*

Coming back to the code, the user is then asked to select the kind of threshold (standard or adaptive, normal or inverted), operating directly on both the full image and the master image, according to the illumination conditions and the background colour: for example, for grey pieces on a white, well-illuminated background, the best choice is standard inverted binary thresholding, while a grey or black background with non-homogeneous illumination could require a normal adaptive thresholding.
The two thresholding values the user has to adjust, called THRESHOLD and ADAPTIVE in Figure 4.8, are the parameters characterizing the functions cv.threshold() and cv.adaptiveTreshold().
Of course, these two functions cannot be applied at the same time.

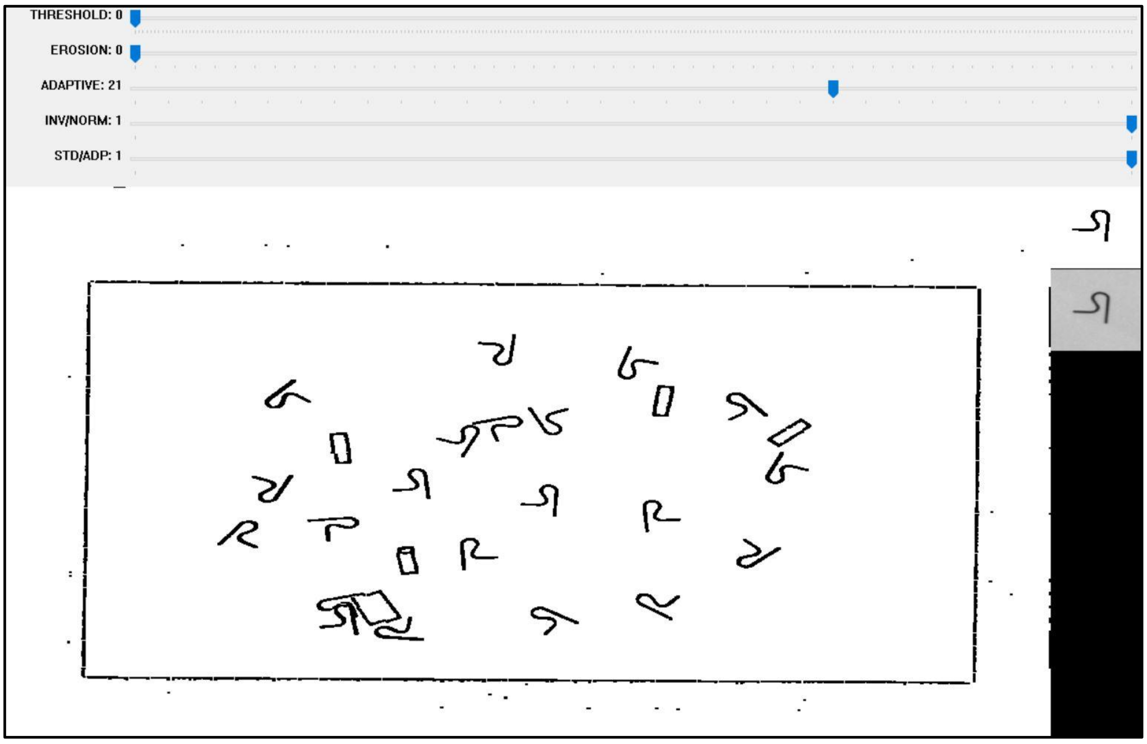

THRESHOLD: 0
EROSION: 0
ADAPTIVE: 21
INV/NORM: 1
STD/ADP: 1

***Figure 4.8**: thresholding settings window. By regulating the five trackbars (top), the user imposes the parameters for the thresholding of the full image (centre) and of the master (top right). The results can be compared in real time with the master itself (bottom right)*

Function 4.1*: cv.threshold() [28]*

- Input:
    - **src**: source greyscale image.
    - **thresh**: the THRESHOLD parameter value, from 0 to maxval.
    - **maxval**: maximum value to use with binary and inverted binary, usually set to 255.
    - **type**: cv.THRESH_BINARY or cv.TRESH_BINARY_INV, for this work, but it can accept other kinds of thresholding.

- Output:
    - **dst**: output greyscale image.

Function 4.2*: cv.adaptiveThreshold() [29]*

- Input:
    - **src**: source greyscale image.
    - **thresh**: THRESHOLD value, from 0 to maxval.
    - **maxval**: maximum value, usually set to 255.
    - **adaptiveMethod**: mean-based or gaussian-based adaptive thresholding. For this work it has been chosen the gaussian method cv.ADAPTIVE_THRESH_GAUSSIAN_C since it results less sensitive to noise.
    - **blockSize**: the ADAPTIVE parameter value, it is the size of pixel neighbourhood used to calculate the local threshold. It can assume only odd values equal or greater than 3.
    - **C**: constant subtracted from the weighted mean; it is set to 5 but it can assume any value.

- Output:
    - **dst**: output greyscale image.

To have a better vision of the different results when the two functions are adopted on the same image, Figure 4.9 shows the resulting adaptive and standard thresholding applied to another kind of pieces, tagged as diapason188.

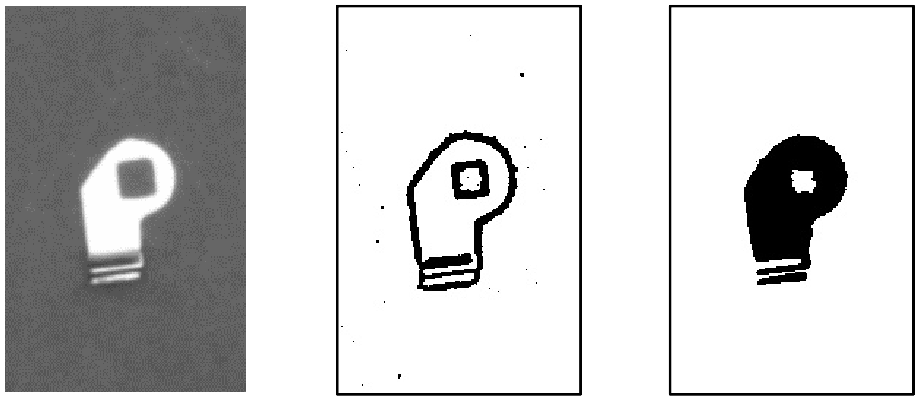


***Figure 4.9**: adaptive and standard (global) thresholding on diapason188. From left to right: master, adaptive thresholding result, and standard (global) thresholding result*

However, selecting the correct set of thresholding parameters could not be enough to obtain a good result, since the user might occur in two problems: noise in the form of randomly distributed black pixels and non-uniformity of the blob.
The solutions for both of those issues are based on a kind of techniques called morphological operations, three of those are used for this work (Figure 4.10):

- Dilation: given a kernel, (usually squared), it dilates the white areas. In a black-and-white image the operation results to be very simple: if the kernel is, for example, a 3X3 square, the eight pixels surrounding each white pixel are turned into white.

- Erosion: inverse of dilation, where the black areas are expanded and white pixels surrounding a black one, are turned into black.

- Closing: dilation followed by erosion to eliminate unwanted black points on a white background. Usually, those black or white isolated points are the result of the thresholding applied to the background noise.

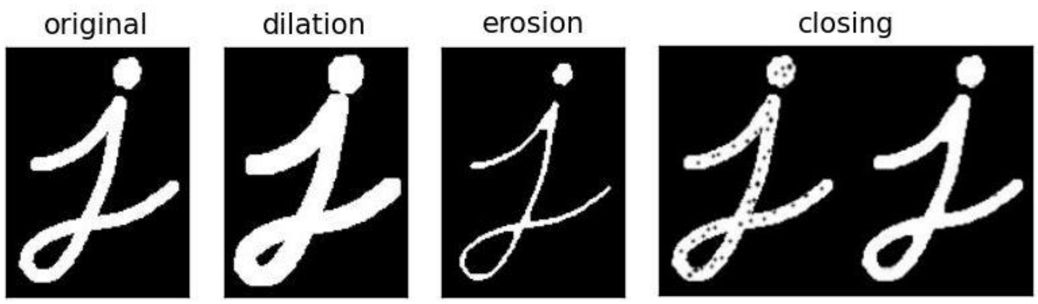


***Figure 4.10**: bitwise operations examples [30], [31]*

To solve the noise issue, the program applies a closing operation with a fixed 5X5 square kernel, while, to solve the non-uniformity of the blob, an additional user operation is required.
The non-uniformity of the blob is a significant issue for the contour detection phase and may occur when the piece is not planar and reflects the source light in different ways or when the piece is characterized by holes or protrusions. Usually, this problem is automatically solved when adaptive thresholding is adopted: however, it may happen that the user must impose standard thresholding because of illumination conditions, obtaining multiple isolated blobs. The solution is a simple erosion operation where the number of iterations is set to 1 and the user selects the kernel size by adjusting the EROSION parameter (Figure 4.11).

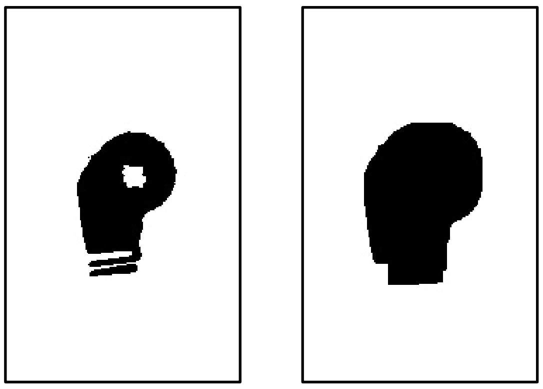


***Figure 4.11**: erosion and closing. By applying erosion and closing on the result of standard thresholding, the noise effect is eliminated and the blob appears as a uniform spot*

To summarise, given the full image and the master, the user must select the kind of thresholding (standard/adaptive, normal/inverted) to properly separate the objects from the background, then, if needed, he must unify multi-blobs by setting the erosion kernel sizes.
The result is the final segmentation, shown in Figure 4.12.

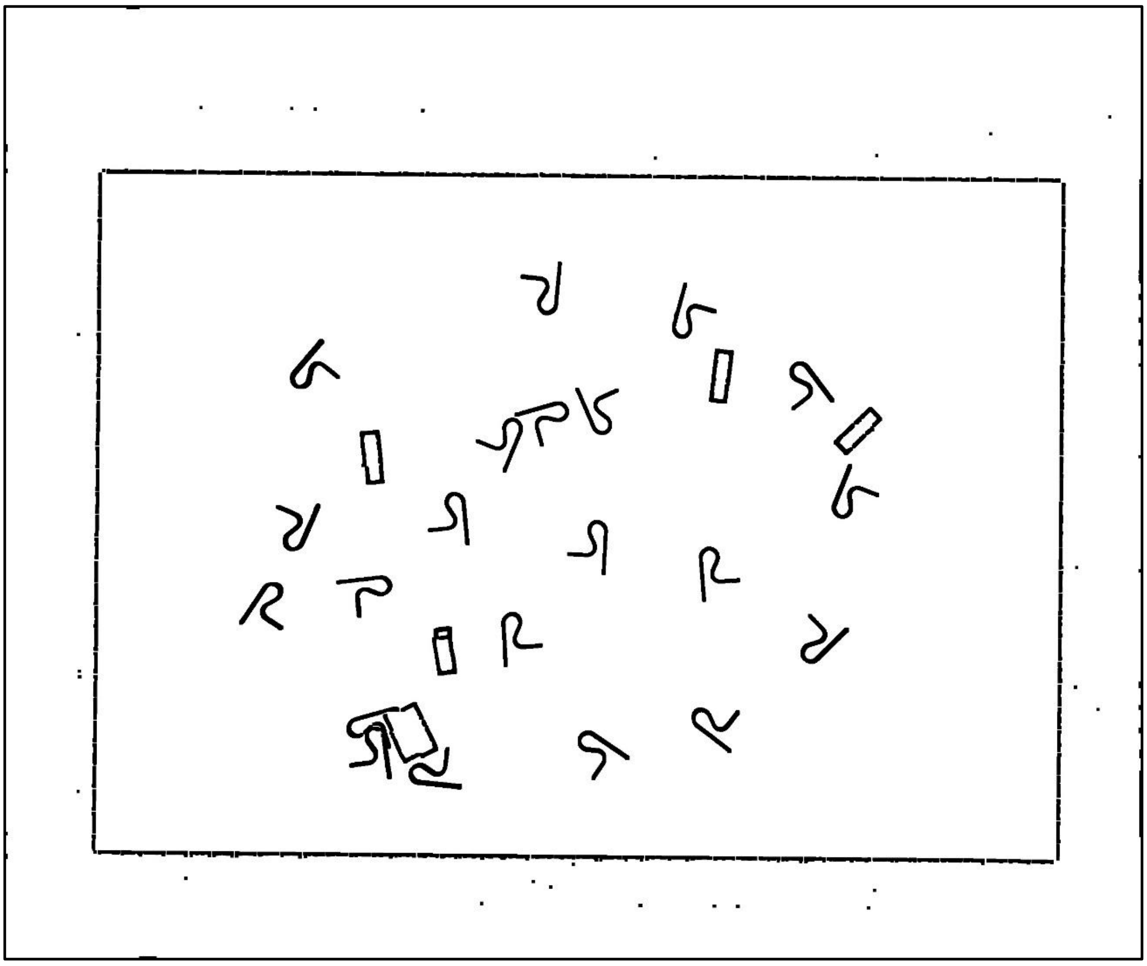


Figure 4.12: final segmentation. On the full image a normal (not inverted) adaptive thresholding with ADAPTIVE parameter equal to 21 has been applied

4.6 Contours, areas, and centroids

Now that the segmentation phase is completed, from Figure 4.12 it is possible to notice how some blobs correspond to wrong-posed pieces or to huge groups of pieces too close to each other to be considered by the robot.
The next step consists then into excluding those blobs by computing the area of the master blob and, fixed a tolerance range, selecting the blobs from the full image whose area is included in that range.

To find the area of each blob it is mandatory to find the contour first, starting with the master: the contour, i.e. the curve that joins all the continuous points along the blob boundary, is detected by the program with the function cv.findContours() [32] but it is selected by the user (Figure 4.13), since there could be more than a single detected contour if the piece presents holes or big zones with high intensity gradient, as shown in Figure 4.14.

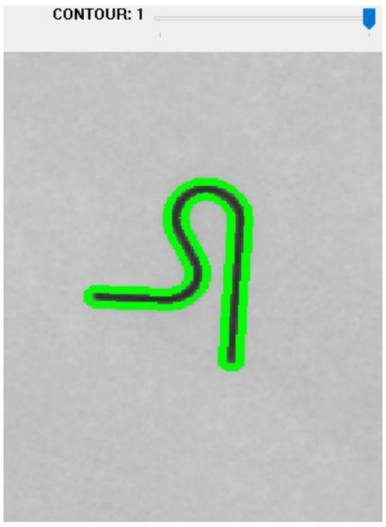


***Figure 4.13**: master contour selection window*

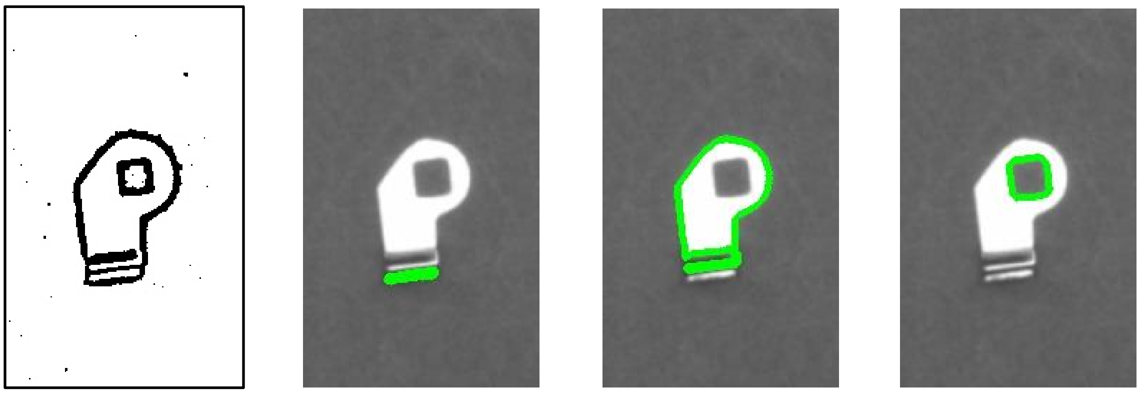


***Figure 4.14**: diapason188 multiple contours*

The possibility of choosing the contour makes it possible, in many cases, to avoid the erosion operation seen before: in the example shown in Figure 4.14, the blob (obtained with adaptive thresholding) is characterized by three different contours (green curves). However, it can clearly be seen that the second one is enough to effectively describe by

itself the whole piece shape: erosion would have been mandatory if the three contours had significant importance in the definition of the piece shape.
Given the master contour, its area and momentum are computed with cv.contourArea() [33] and cv.moments() [34]: the area is fundamental for the pieces blobs detection, while, from the momentum, it is possible to compute the coordinates of the centre of mass (centroid) of the blobs, indicated as c_x and c_y. That is another reason why the contour must be as uniform and well-shaped as possible, since too different contours from the ideal shape of the piece would lead to centroid misplacing.

At this point, before proceeding with areas comparison, as a matter of precision the program crops a new master from the original full image, identical in size and orientation to the previous one, but the image centre is now coincident with the centroid of the blob: this is the definitive master image that is going to be saved as a reference for the entire process.
In Figure 4.15, it is possible to notice a very little difference between the two masters: this implies that the user must not be extremely precise with the initial master cropping, since it will centre itself at this point of the process.

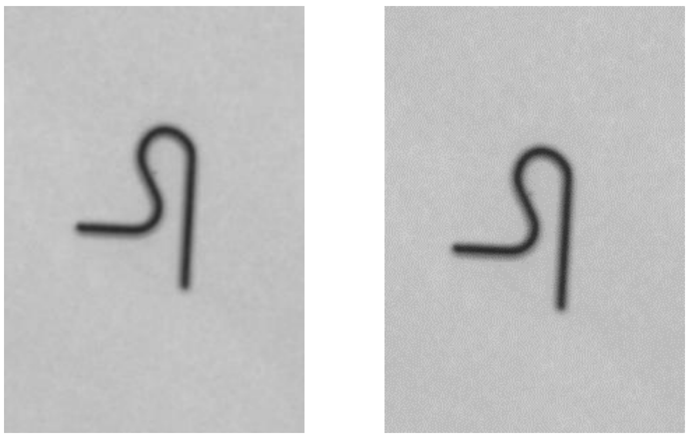


Figure 4.15: *definitive reference (master) image. The new master image (right) is simply a new version of the master image cropped by the user whose centre coincides with the centre of mass of the master blob*

4.7 Candidate objects identification

This is the last candidate objects identification step, now that all the contours and the master blob area are available. The procedure can be schematized as:

1) The generic blob area from the full image is computed from the blob contour with cv.contourArea().

2) A tolerance range, for example $\pm 80\%$ of the master blob area, is considered.

3) The generic blob area is compared with the tolerance range: if it is included, the corresponding object is considered a candidate object, otherwise it is ignored.

4) The generic candidate object centroid coordinates c_x and c_y are computed.

5) c_x and c_y are compared with three sets of coordinate limits, the last two of which and their functions will be defined in detail in the next chapters:

    - Grip constraints, defined at the beginning of this chapter, to be sure that the centroid falls in the safety area for the robot.

    - Crop limit, to guarantee that the target image for the feature matching phase can be cropped from the full image.

    - Grip limit, to guarantee that the collision avoidance area can be defined, no matter the rotation angle of the piece.

    If all the three checks are passed, the centroid and its corresponding piece are collected in the candidate objects set.

This procedure takes place every time the camera takes a new picture of the platform, and the result is printed on screen: as shown in Figure 4.16, the program shows all the detected contours (green), and the candidate objects by drawing their centroids (red), their minimum area rectangle (blue), and the Ids that characterise the objects from now on (blue).

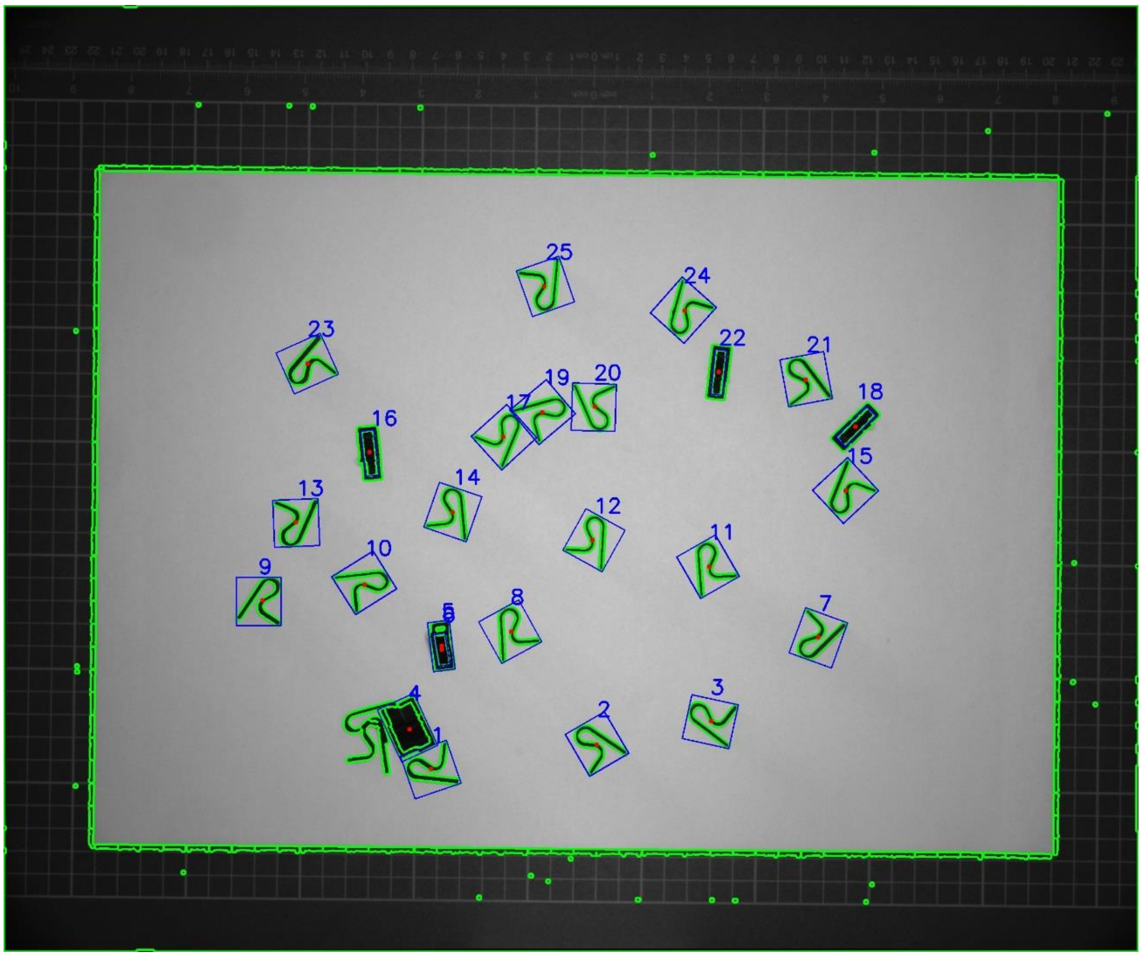
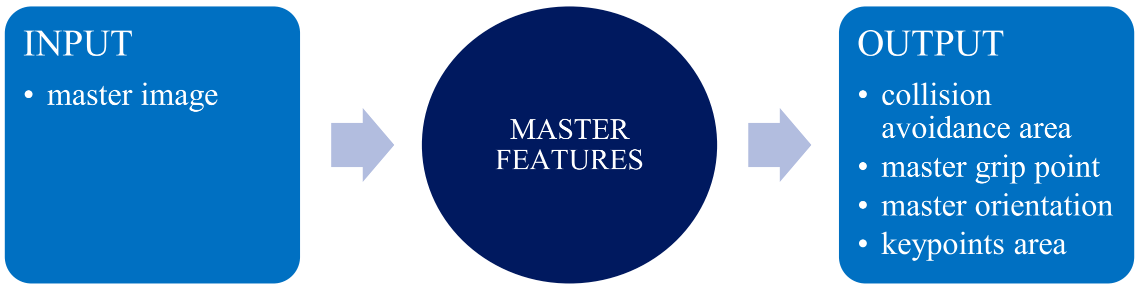


***Figure 4.16**: candidate object detection on the full image*

Chapter 5: master features

5.1 Introduction

This is going to be a quite short chapter, since it concerns the definition of all those parameters that must be determined by the user before proceeding with feature matching: collision avoidance area, master grip point, master orientation, and keypoints area.



As for the master thresholding parameters and the master blob area, all these values must be set only the first time a new kind of pieces is considered.

5.2 Collision avoidance area

The first operation on the master image is the definition of the collision avoidance area, i.e., the area that must result obstacle-free to consent the grip by the robot.
It is defined by the user as the area in-between two polygons: an inner one (blue) is first determined by drawing the vertexes and it must be very close to the piece, then, once the

inner polygon is closed, the user must trace also the outer one (red) according to the grip characteristics of the robot (Figure 5.1).

In this phase it is also computed the maximum distance between the centre of the master image and the vertexes of the outer polygon: if the distance between the centre of a candidate piece blob and one of the four grip constraints of the full image is lower or equal to that maximum distance, the piece is automatically excluded. This operation is performed to guarantee the possibility to crop for each candidate object an image large and high enough to host the red polygon, no matter its rotation around the centroid.
This limit, called grip limit, is one of the two limitations considered at the end of the previous chapter.
The method adopted to determine if that area is obstacle-free or not is based on the Canny edge detection method, explained in detail in Chapter 7.

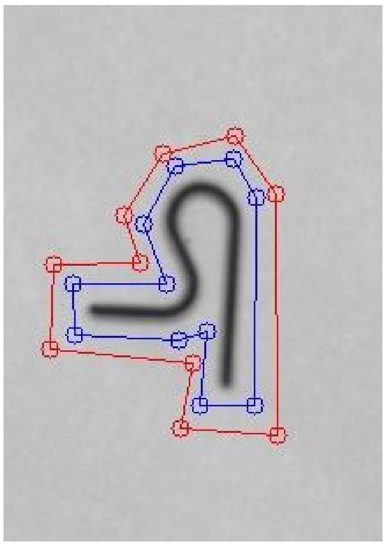


Figure 5.1: *collision avoidance area*

5.3 Grip point and master orientation

The next simple but crucial operations are the definition of the reference grip point and the orientation of the master: the user indicates a point on the master and defines an arrow with origin in the centre of the image, indicating the orientation of the master with respect to the horizontal (Figure 5.2).

For the grip point, the program computes the distance from the centre of the image and the angle between this segment and the horizontal, while, for the arrowhead, since its length has just representation purposes, it is enough to compute the angle.
All those parameters are necessary to the computation of the grip point cartesian coordinates of the generic piece and its orientation.

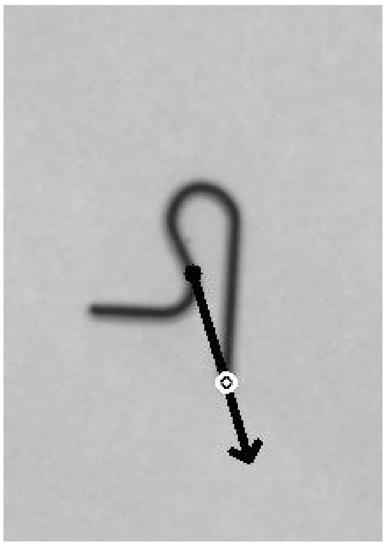


Figure 5.2: *master grip point and orientation arrow*

5.4 Keypoints area

Keypoints are the main protagonists of the feature matching process, that will be largely discussed in the next chapter. Just to introduce them in a very synthetic way, keypoints are interesting points in an image detected by a specific algorithm and compared to other keypoints of another image: if two keypoints from two different images have similar characteristics, the program matches them with another dedicated algorithm.
Nevertheless, it may happen that just a limited area of the master image has significant characteristics for the comparation: for this reason, the user can trace a third polygon (Figure 5.3) to consider only the keypoints included in that area, gaining a lot of computational time in the next phases.

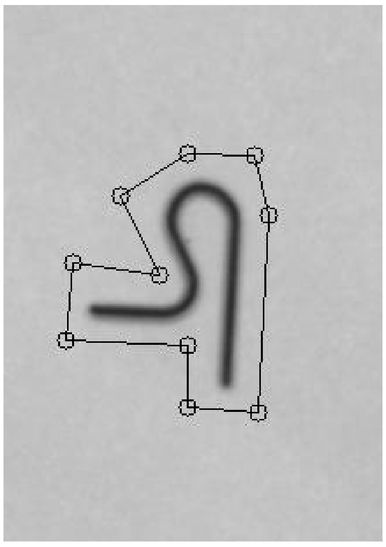
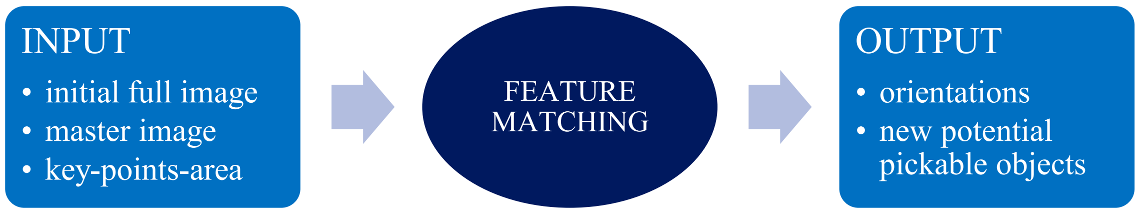


***Figure 5.3**: keypoints area*

Chapter 6: feature matching

6.1 Introduction

This chapter represents the main core of the whole project since it is about the part of the program that compares each candidate object with the master, finds its relative orientation angle, and performs a further control to exclude more wrong pieces. The basic concept behind this procedure is feature matching.



Again, this chapter will follow the code order, presenting at first a brief theoretical description of two features detectors and descriptors called SIFT and ORB, and two matchers called Brute-Force (BF) and FLANN, in additional to homography for what concerns SIFT. Then, it will describe the procedure to obtain the pieces orientation as the difference angle between each piece rotation and the master rotation.

6.2 Features

To better understand what features are and how feature matching works, it is better to introduce them with an example. Let's consider Figure 6.1 as a very simple game where the player must find in the full picture the objects represented in the small rectangular windows, going from A to F.

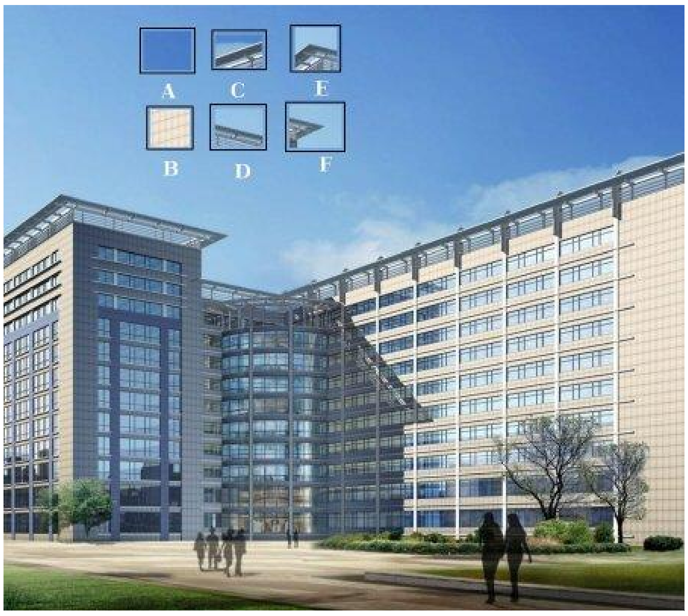


***Figure 6.1**: features example [35]*

To accomplish this simple quest, the human brain, through the eyes, looks for relevant characteristics in each window and in the full image as changing in colour, light gradients, or peculiar shapes. Those characteristics are the so-called features.
Then, once the features have been detected, the brain elaborates a description of each feature and looks for similar descriptions in the windows and in the full image: if two features have same characteristics, they are matched.

This, in program language, is feature matching. The program is then composed of:

1) Detectors: algorithms that look for features in an image. The more these features are characterized, the easier and more precise is the tracking: for example, corners, edges, or blobs are good features, while flat areas are bad features.

2) Descriptors: algorithms that look for the features found by the detector in another image. SIFT and ORB are both detectors and descriptors.

3) Matchers: algorithms that match similar features of different images, like Brute-Force and FLANN.

6.3 SIFT

SIFT (Scale Invariant Feature Transform) is an algorithm that detects keypoints and computes their descriptors proposed in 2004 by David Lowe in his paper [36]. The keypoints detected and described by SIFT result to be robust and matchable to large datasets of objects. Also, many keypoints can be generated for small objects, like the master image obtained in the previous chapters.
To briefly explain the algorithm procedure, it has been followed the 5-steps explanation provided by OpenCV, based on the Lowe paper organization:

1) Scale-space extrema detection: the main issue with simple detectors as Harris or Shi-Tomasi is that they are rotation invariant (the corner is found no matter the rotation) but not scale invariant, since it is not possible to use the same window with constant sizes to detect keypoints with different scales.
   For this reason, SIFT at first separates scale-space into several octaves forming a Gaussian pyramid, where octaves number depends on the original image size and each octave size is half of the previous one. Then, a scale-space filtering approach is adopted by computing Difference of Gaussians (DoGs) as the difference of Gaussian blurring of an image with two different scale values σ and kσ, where σ is the scaling parameter. This process is done for different octaves in the pyramid (Figure 6.2).

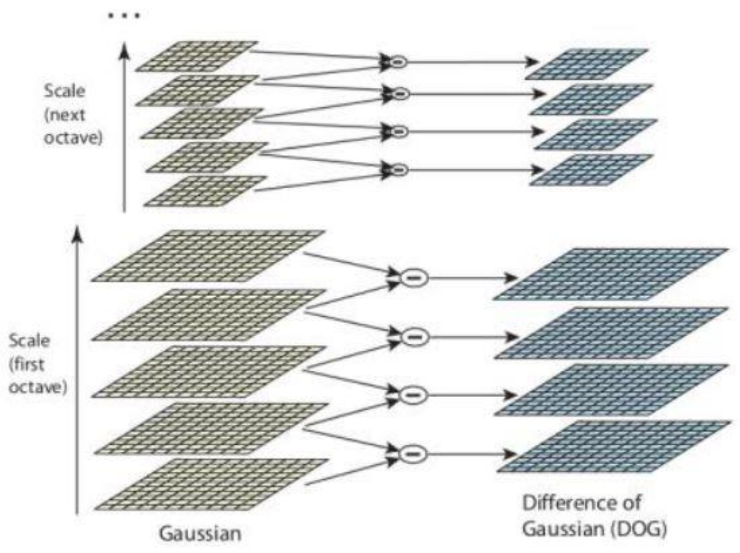


***Figure 6.2**: DoG on different image octaves [37]*

Once the DoGs are computed, a pixel in an image is compared with its 8 neighbours and the 9 pixels in the next and previous scale (Figure 6.3): if the pixel is a local extremum, it means that its scale better represents it, and it is considered as a potential keypoint.

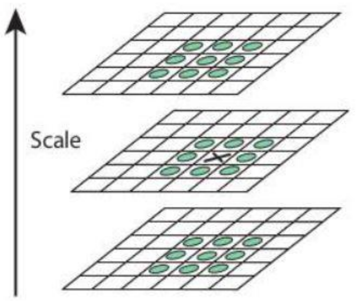


***Figure 6.3**: keypoint identification [38]*

2) Keypoint localization: some keypoints resulting in the previous step as low contrast keypoints and edge keypoints are of weak interest: for this reason, potential keypoints locations are refined using Taylor expansion for a more accurate result, excluding all the keypoints with intensity lower than a threshold (contrastThreshold).

   Because of the strong DoGs response for edges, the Hessian matrix H is computed for each keypoint to obtain the principal curvatures since, for weak keypoints, the principal curvature across the edge is much larger than the principal curvature along it. The parameter that takes into account this situation is defined by the ratio between the squared trace and the determinant of H:

$$R = \frac{Tr(H)^2}{Det(H)} \tag{6.1}$$

   If R is greater than a threshold (edgeThreshold), the keypoint is excluded.

3) Orientation assignment: to guarantee rotation invariance, for each keypoint gradient intensity and direction in its neighbourhood are computed. Then, a 36-bin histogram is created, covering 360°: the highest histogram peak and any other peak above 80% of the highest peak are considered to compute orientation.

However, since there could be several peaks, several keypoints can be created with same location but different orientation, affecting matching stability.

4) Keypoint descriptor: for each keypoint, the descriptor is created as a 128-bin vector, obtained from a 16X16 neighbourhood around the keypoints, divided into 4X4 sub-blocks from which 8-bins histogram is created.
   The descriptor results to be highly distinctive and invariant as possible to external variations as illumination or perspective changes.

5) Keypoint matching: two keypoints are matched by identifying their nearest neighbours. However, it may happen that the 2nd closest match is very close to the 1st because of noise or other reasons: in this case, if the ratio between the 1st closest distance and the 2nd is greater than a threshold, the match is rejected: this procedure is called Lowe ratio test, since it is presented by Lowe in its paper where, with a threshold of 0.8, it eliminates the 90% of false matches with a waste of only 5% of correct matches (Figure 6.4).
   However, for this work the threshold has been fixed to 0.75 as a compromise between the value proposed by Lowe and the minimum value of 0.7 proposed by OpenCV for the matching tutorials.

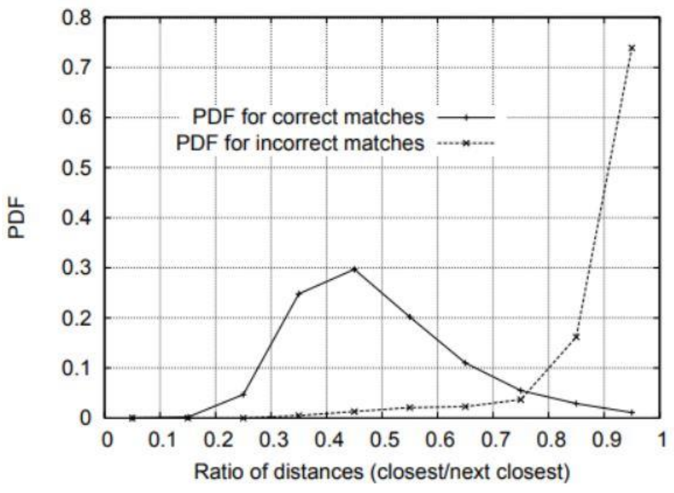


Figure 6.4: *Lowe ratio test [39]*

This test will be performed for both SIFT and ORB method, independently from the kind of matcher algorithm adopted.

6.4 ORB

ORB (Oriented FAST and Rotated BRIEF) is an algorithm proposed in 2011 by Ethan Rublee, Vincent Rabaud, Kurt Konolige and Gary R. Bradski in their paper [40] and developed at OpenCV Labs that results to be a good alternative to SIFT and SURF (SIFT was patented in 2011 so not free) in terms of computation cost and matching performance. As the name says, it is based on two algorithms for keypoints detection and description, by improving their performances:

- FAST (Features from Accelerated and Segmented Test) detector: every pixel p in an image is compared with its 16 neighbours in a circular area that are classified as brighter, darker, or similar to p. If more than 8 pixels are darker or brighter, p is a keypoint.

- BRIEF (Binary Robust Independent Elementary Feature) descriptor: it takes all the keypoints from the detector and converts them into a binary feature vector. Each vector represents a keypoint and it consists in a binary 128-512 bits string.

At first, ORB creates a pyramid, similarly to what SIFT does as a multiscale representation of the image, where each scale has a lower resolution. Then, it applies FAST detector to find keypoints, followed by Harris corner measure to select the top N among them.
However, FAST doesn't provide orientation so, a modification is required: for each keypoint it is computed the intensity weighted centroid of the patch with located corner at centre. The orientation is the direction of the vector that goes from the corner point to the centroid, while the rotation invariance property comes from the computation of the moments of the patch.

Once the keypoints have been detected, they need to be described by the BRIEF algorithm that poorly performs with rotation, so another modification is required.
First, the image is smoothered with a Gaussian kernel to prevent unwanted high-frequency noise effect. Then, to maintain the BRIEF characteristics, ORB 'steers' BRIEF that creates a vector and a patch for each keypoint and performs a set of n binary tests, whose binary vector result defines a feature. Then, for any feature set of n binary tests it is defined the 2Xn matrix S, containing the pixel coordinates. S is then rotated using the orientation angle and obtaining the rotated matrix S_θ, used to compute the keypoint descriptor.

However, one of the issues with steered BRIEF is that each bit feature loses the characteristic property of having a large variance close to 0.5 that now becomes more distributed, and the feature becomes less discriminative. To solve this problem and to guarantee the test uncorrelation at the same time, ORB runs each test again for all the patches and orders them by their distance from the ideal value of 0.5 forming the vector T. At this point, ORB creates a vector R in which, at first, the first test of T is inserted and, at the same time, removed from T. The next T test is then removed from T and compared with all the R tests: if its absolute correlation is lower than a threshold it is added to R, otherwise it is discarded. This procedure is repeated until R contains 256 tests: if at the end there are less than 256 tests, the threshold is raised, and the procedure starts again. This method is called rBRIEF.

As for matching, the main difference with SIFT is the use of a multi-probe LSH that makes this algorithm faster than SIFT that adopts a traditional LSH.

6.5 Matchers

Matcher algorithms are responsible for the matching between the training set of keypoints from the first image and the target set of keypoints from the second one.
This paragraph introduces the two matchers proposed by OpenCV, i.e., Brute-Force and FLANN:

- Brute-Force (BF): it is the simpler, since for each keypoint of the first set, its descriptor is matched with all the descriptors of the second set, returning the closest one. The computation is based on the Euclidean distance between the keypoints when it is applied to SIFT, while it is based on the Hamming distance when it is applied to ORB.

- FLANN: Fast Library for Approximate Nearest Neighbours, contains a collection of algorithms optimized for large datasets and high dimensional features.

To obtain more accurate results, it is also possible to apply a technique called homography, very common when the goal is to obtain the perspective transformation of an object. Even if we are not dealing with perspective, given the fact that pieces on the platform must have the same pose of the master to be considered, this technique can be useful anyway, since it implements algorithms like RANSAC that estimate the matches correctness, collecting only the good ones (inliers) and excluding the bad ones (outliers). In this project, homography will be applied only to the combinations SIFT+FLANN and ORB+FLANN.

6.6 Code

As already mentioned in the introduction, the purpose of this part of the code is to obtain the orientation of the pieces in terms of rotation angle with respect to the master piece orientation, imposed by the user in the last chapter.

The feature matching code is divided in two main parts: the first one is on the master side, and it must be performed just one time for each kind of pieces, like all the other master operations. The first passage of this phase is to obtain the master keypoints by creating the required class with the OpenCV functions cv.SIFT_create() or cv.ORB_create() and setting all their characteristic parameters. The choice of some of those parameters is not trivial and it depends on lots of factors as illumination conditions, background colour, and camera resolution: for this reason, those parameters are chosen empirically for each piece, after many trials that are going to be discussed in the tests chapter.

***Function 6.1**: cv.SIFT_create() [41]*

- Input:
  - **nfeatures**: number of features to retain, ranked by their local contrast score.
  - **nOctaveLayers**: number of layers in each octave, here set to 5.
  - **contrastThreshold**: threshold to reject weak features in low-contrast regions. It must be decreased to detect more keypoints.
  - **edgeThreshold**: threshold to reject edge-like features. It must be increased to detect more keypoints.
  - **sigma**: sigma corresponding to the octave 0 of the input image. Here it is fixed at its optimal value 1.6 but it can be increased in case of weak camera with soft lenses.
  - **enable_precise_upscale**: Boolean value. If true, prevents localization bias.
- Output:
  - **sift**: class SIFT.

Function 6.2*: cv.ORB_create() [42]*

- Input:
  - **nfeatures**: maximum number of features to retain.
  - **scaleFactor**: pyramid decimation ratio, always > 1 but it must be nor too high (dramatic degradation of feature matching score), neither too close to 1 (computation speed would decrease). Here fixed to the 1.2 default value.
  - **nlevels**: number of pyramid levels, here fixed to 10.
  - **edgeThreshold**: size of the border where features are not detected.
  - **firstLevel**: pyramid level corresponding to the source image, here fixed to the default value 0.
  - **WTA_K**: number of points produced by each element of rBRIEF descriptor, here fixed to the default value 2.
  - **scoreType**: algorithm used to rank features. Here it is used the default HARRIS_SCORE, corresponding to the 0 value.
  - **patchSize**: size of the patch used by rBRIEF.
  - **fastThreshold**: threshold characterizing FAST detector.

- Output:
  - **orb**: class ORB.

Once the algorithm class is defined, master keypoints and their descriptors are obtained with sift.detectAndCompute() or orb.detectAndCompute() and saved in a file to be used for every future application with the same pieces.

Function 6.3*: sift.detectAndCompute() [43]*

- Input:
  - **image**: greyscale image.

- Output:
  - **kp**: KeyPoint class [44], whose accessible attributes are pt (coordinates of the keypoints), size (neighbourhood diameter), angle (rotation with respect to the horizontal), response (by which the strongest keypoints have been selected), octave (pyramid layer), and class_id (object class).
  - **des**: descriptors.

***Function 6.4**: orb.detectAndCompute() [45]*

- Input:
  - **image**: greyscale image.
- Output:
  - **kp**: KeyPoint class [46], whose accessible attributes are pt (coordinates of the keypoints), size (neighbourhood diameter), angle (rotation with respect to the horizontal), response (by which the strongest keypoints have been selected), octave (pyramid layer), and class_id (object class).
  - **des**: descriptors.

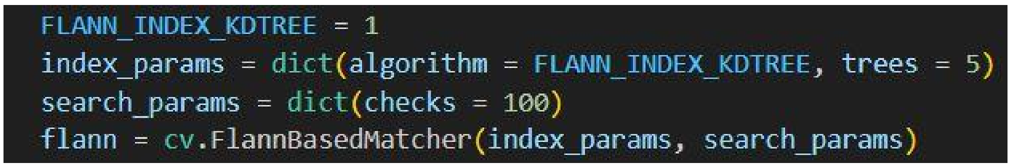
At this point, all the operations on the master are performed and every line of code from now on is going to be repeated every time the program runs.
On the target side, the procedure is quite similar for what concerns the keypoints detection and description, since the class type and its parameters are the same: the additional step is the definition of the matcher type with cv.BFMatcher.create() or cv.FlannBasedMatcher().

***Function 6.5**: cv.BFMatcher.create() [47]*

- Input:
  - **normType**: NORM_L1 or NORM_L2 for SIFT, NORM_HAMMING or NORM_HAMMING2 for ORB.
- Output:
  - **bf**: class BF.

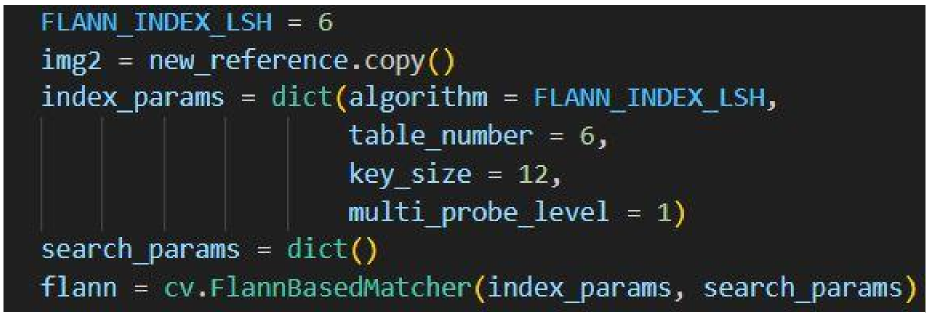
***Function 6.6**: cv.FlannBasedMatcher() [48]*
• Input: o **indexParams**: dictionary to specify the algorithm to be used, it depends on the detector/descriptor. o **searchParams**: dictionary to specify the number of times the tree in the indexParams dictionary should be recursively traversed. High values lead to better results but are more time consuming. • Output: o **flann**: class FLANN.

The initialization code lines for cv.FlannBasedMatcher() are shown in Figure 6.5 for SIFT and in Figure 6.6 for ORB.



```
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)
search_params = dict(checks = 100)
flann = cv.FlannBasedMatcher(index_params, search_params)
```

***Figure 6.5**: FLANN initialization for SIFT*



```
FLANN_INDEX_LSH = 6
img2 = new_reference.copy()
index_params = dict(algorithm = FLANN_INDEX_LSH,
                    table_number = 6,
                    key_size = 12,
                    multi_probe_level = 1)
search_params = dict()
flann = cv.FlannBasedMatcher(index_params, search_params)
```

***Figure 6.6**: FLANN initialization for ORB*

However, before starting with the detection it is necessary to prepare the target images: for each candidate object detected at the end of Chapter 4, the program crops two images centred in the object centroid but with different sizes. At this point the meaning of the so-called crop limit and grip limit (also imposed at the end of Chapter 4) becomes clear: the centroid of a generic candidate object must be far enough from the borders of the full image so that the target image relative to that piece can be cropped, having the same size of the master (crop limit) and also it must be possible to crop a second image with twice the size of the grip limit to guarantee that every point of the collision avoidance area polygons are contained in the cropped image, no matter the rotation. The first one of these two images is used as target in the matching phase, while the second one will be used in the next chapter for the Canny edge detection method application.

Once the target image is cropped, the target keypoints are detected and described with the same functions and the same parameters used for the master, while the matches are obtained and collected with bf.knnMatch() or flann.knnMatch(), but only if the master keypoint of a pair is contained in the keypoint area defined in Chapter 4.

***Function 6.7**: bf.knnMatch() [49]*
• Input: o **queryDescriptors**: des from the target image.   o **trainDescriptors**: des from the reference image.   o **k**: count of best matches found per each query descriptor. Fixed at 2. • Output: o **matches**: each element is k or less matches for the same query descriptor.

***Function 6.8**: flann.knnMatch() [50]*
• Input: o **queryDescriptors**: des from the target image.   o **trainDescriptors**: des from the reference image.   o **k**: count of best matches found per each query descriptor. Fixed at 2. • Output: o **matches**: each element is k or less matches for the same query descriptor.

The retained matched keypoints are then converted into numerical coordinates to make them easily readable by the program and the user and the Lowe ratio test is applied to eliminate all those pairs for which the ratio between the Euclidean distances of the two keypoints is less than 0.75. If the Lowe ratio test is passed, the match is classified and collected as ‘good’.

If the matching algorithm also considers the homography application, it is implemented at this point with the function cv.findHomography(), that provides a binary mask to apply to each pair where 0 means bad matching and 1 good matching.

***Function 6.9**: cv.findHomography() [51]*
• Input: o **srcPoints**: coordinates of the points on the target image. o **dstPoints**: coordinates of the points on the reference image. o **method**: method used to compare homography, here it is always used RANSAC. o **ransacReprojThreshold**: maximum allowed reprojection error to consider a point pair as inlier. It is usually taken between 1 and 10, here is fixed to 7. • Output: o **mask**: output inliers mask.

Before starting to work with angles, all the pairs are subject to a multi-check condition that classifies and collects a pair as ‘very good’ if:

- The homography mask (if present) is 1.

- The difference of the distances between the two keypoints and the centre of their own corresponding image is less than a small number of pixels, 10 in this project, since two matched keypoints should be ideally placed at the same distance from the centroid of the piece. This is done to avoid huge mismatches that may happen when in the target image is also present a second piece, or part of it, and the matcher considers the keypoints from this piece instead of the keypoints of the desired main one.

To practically visualize what has been done until now, Figure 6.7 shows the result of a well-posed piece (21st piece, referring to the full image in Figure 4.16) with the target on the left and the master on the right. The keypoints and the ‘very good’ match pairs are shown with the function cv.drawMatches() [52] that represents the keypoints as a circle with a segment inside, where the circle size indicates the keypoint size and the segment indicates the keypoint orientation, while the matches are represented by coloured lines. Of course, no keypoint outside the keypoints area previously defined on the master is matched.

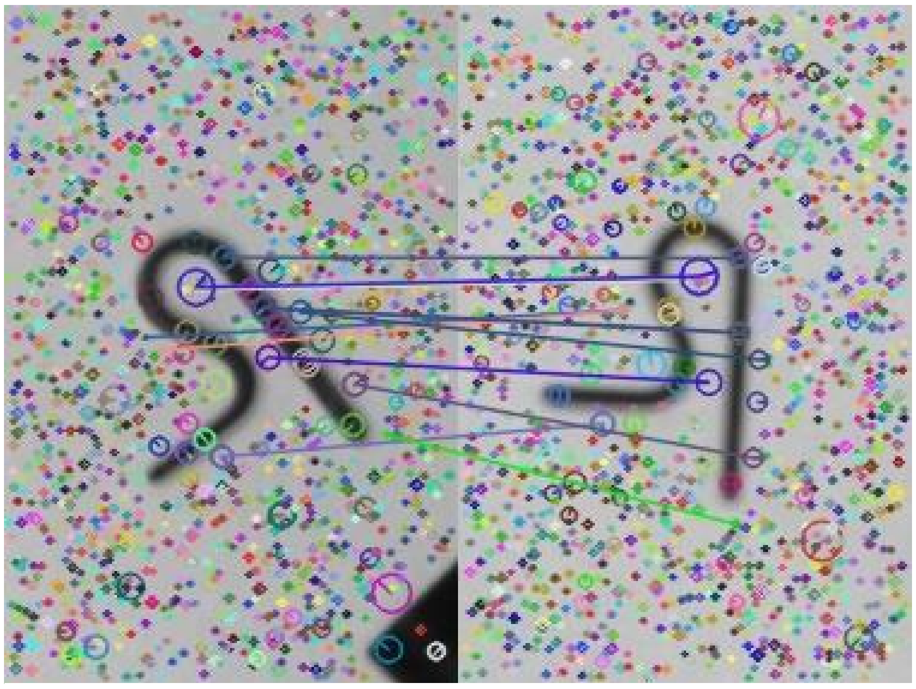


***Figure 6.7**: matching result on a well-posed piece. The applied algorithm is SIFT+FLANN+H*

This passage results to be a very important check since it already discards lots of wrong pieces: as shown in Figure 6.8, if a piece has completely wrong pose, the number of pairs found is very low or even zero. It is then possible to ignore wrong pieces by simply imposing a minimum number of matched pairs that must be classified as ‘very good’.

Anyway, the control on the ‘very good’ matches number is still not sufficient to discard all the wrong posed pieces, since it may happen that a minimum number of ‘very good’ matches is found, as in cases of symmetry conditions as also shown in Figure 6.8. This issue will be automatically solved with the check on the collision avoidance area in the next chapter.

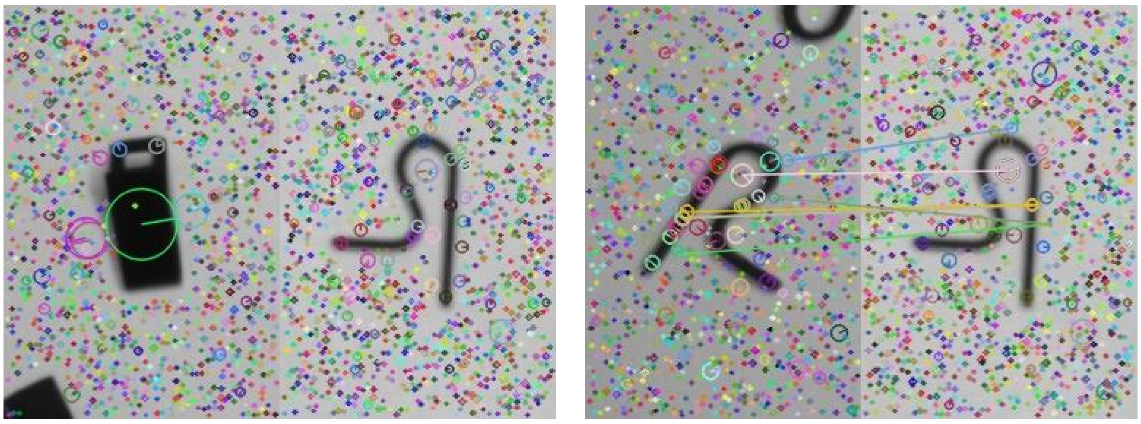


Figure 6.8: *matching result on wrong-posed pieces. The target piece on the left (5th in the full image) has no match pairs, while the one on the right (the 9th) presents match pairs even if its pose is incorrect*

If both checks are passed, the angles of each keypoint in the pair, indicated from now on with subscript 1 for the target and 2 for the master, are obtained from the original result of bf.knnMatch() or flann.knnMatch() and the difference angle θ is computed at first as: $\theta = \theta_1 - \theta_2$, where $-180° \leq \theta_1, \theta_2 \leq 180°$.
However, while dealing with angle difference, this is not sufficient, since a simple algebraic difference does consider the $\pm180°$ convention: for example, if $\theta_1 = -150°$ and $\theta_2 = 70°$, the difference angle is $\theta = -220° < -180°$, while the desired result should have been 140° in the opposite direction.

To correct all those records, a function has been implemented, whose notations refer to Figure 6.9, in which are represented four goniometric circles corresponding to the position of a generic master keypoint pt2 with coordinates x_2 and y_2 in the four quadrants. The computation of the difference angle depends on the position of the target point pt1 with respect to the position (coordinates) of pt2: if pt1 falls in the circular arc between the green points, the standard equation (written in green) is valid, otherwise the corrected equation (written in red) is valid.

$$for \begin{bmatrix} x_2 > 0 \\ y_2 > 0 \end{bmatrix} \; and \begin{bmatrix} x_2 < 0 \\ y_2 > 0 \end{bmatrix} : \theta = \begin{cases} \theta_1 - \theta_2 \\ 360° - |\theta_1 - \theta_2| \end{cases} \tag{6.2}$$

$$for \begin{bmatrix} x_2 < 0 \\ y_2 < 0 \end{bmatrix} \; and \begin{bmatrix} x_2 > 0 \\ y_2 < 0 \end{bmatrix} : \theta = \begin{cases} \theta_1 - \theta_2 \\ -(360° - |\theta_1 - \theta_2|) \end{cases} \tag{6.3}$$

$$-180° \leq \theta, \theta_1, \theta_2 \leq 180° \tag{6.4}$$

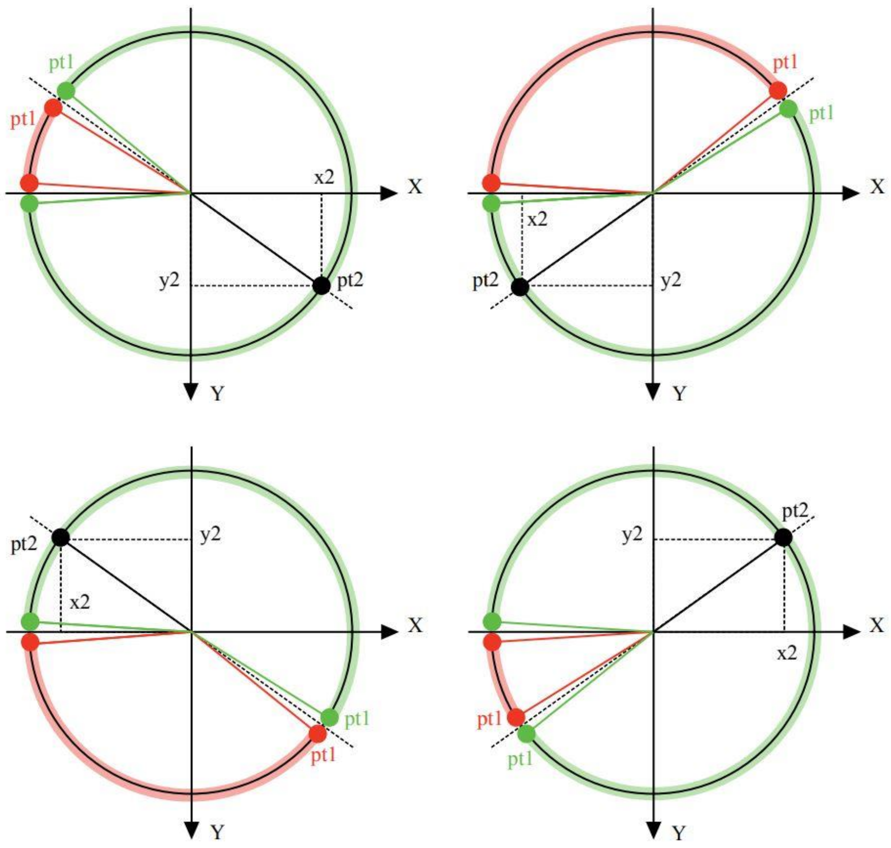


***Figure 6.9**: graphical representation of the difference angles function*

Then, to be sure that the convention is respected, a fast check is carried out:

$$\theta = \begin{cases} 360° + \theta \; if \; \theta < -180° \\ -(360° - \theta) \; if \; \theta > 180° \end{cases} \tag{6.5}$$

This angle function is then applied to every 'very good' match pair and the result is a list of difference angles as long as the number of pairs: the next step is then to obtain from that list a single value assignable to the piece with a simple averaging operation.

To do so, it could be possible to think that it is enough to compute the average of all the remaining difference angles to get the final angle value, but there is one more geometrical issue that could arise.
Let's consider Figure 6.10, which shows a generic master piece on the right and a generic target piece on the left, where the black arrows are the orientations, and the coloured arrows identify three generic pairs of well-matched keypoints.

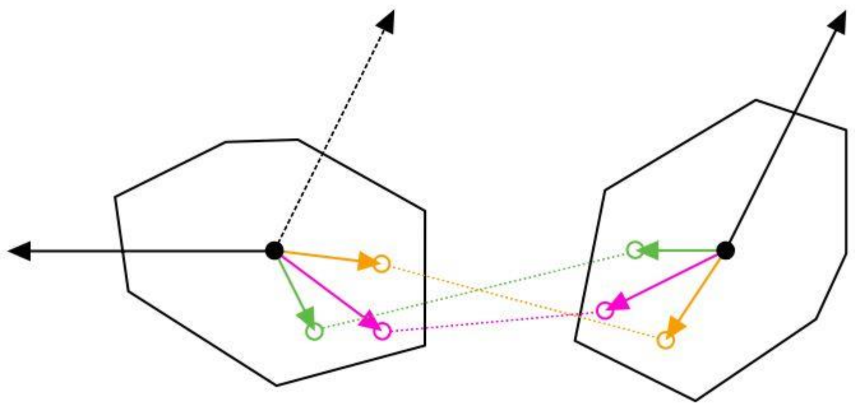


***Figure 6.10**: example of three generic well-matched keypoints*

Now, let's assume that the difference angle θ between the two pieces is somehow known and it corresponds to -120° as in the Figure 6.10: ideally, the difference angles between the three keypoints is expected to be -120° too, but in practice, because of many non-ideal factors as perspective or illumination, the three difference angles assume different values, for example, something like -117.7°, -121.8°, -124.1°.
Anyway, despite those differences, the average of the three angles results to be -121.2°, that is a good approximation, close to the measured -120°.
The problem arises for rotation angles close to 180°: in fact, if the three obtained difference angles are something like 177.7°, -178.2°, -175.9°, the average results to be -58.8° that is a completely wrong result.
To prevent this possibility, the average is only computed by considering the modules of the difference angles, while the sign is assigned to this result with a simple but effective method: if more than a half of the angle signs are positive, the final sign will be positive, otherwise it will be negative.

If the adopted method is a simple averaging, as in this case, it is also necessary to reject all possible outliers before proceeding with the computation, where the outliers are difference angles whose value is very far from the ideal, as result of mismatching due to factors like imperfections on the piece, illumination, and so on.
The adopted method to eliminate them is the IQR method (interquartile range) and consists in the identification of two limits above and below which all the elements of a dataset are eliminated.
In detail, at first the interquartile range (iqr) of the difference angle modules data set is defined as the difference of the 75th and the 25th percentile:

$$iqr = q75 - q25 \tag{6.6}$$

Then, the two limits are computed as:

$$lower\ limit = q25 - iqr * t \quad (6.7)$$

$$upper\ limit = q75 + iqr * t \quad (6.8)$$

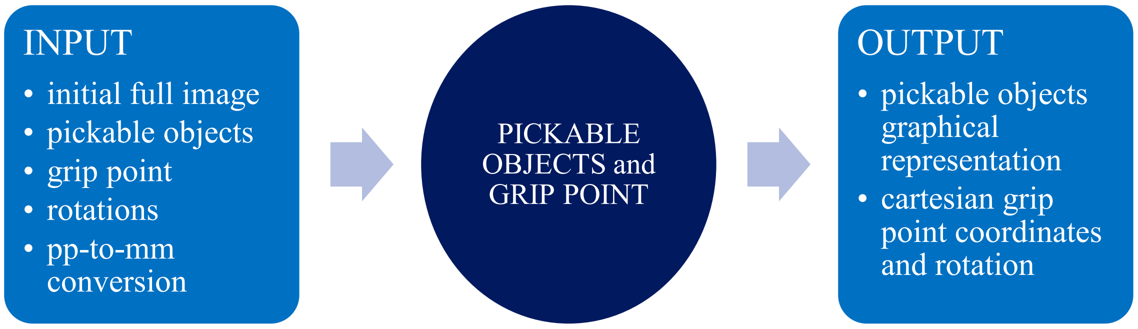
Where t is a threshold empirically fixed at 0.5 for this work.
Once those limits are defined, all the difference angle modules are checked, and a definitive list is obtained: at this point it is enough to average the modules and assign them the sign as specified above to finally obtain the piece orientation.

The next chapter will describe all the operations that lead to the identification of the definitive object list and their grip point coordinates.

Chapter 7: pickable objects and grip point

7.1 Introduction

This chapter describes all controls and checks performed to obtain the final pickable objects list and the coordinates of their grip point.
At the end of the last chapter, a new list of potential pickable objects based on the keypoints matching and their orientation has been obtained: now, the first passage consists in finding the grip point coordinates of each piece and then, by checking the absence of obstacles in the collision avoidance area, drawing up the definitive list of pickable objects, giving to the user their visual representation in the full image.


INPUT
• initial full image
• pickable objects
• grip point
• rotations
• pp-to-mm conversion
PICKABLE OBJECTS and GRIP POINT
OUTPUT
• pickable objects graphical representation
• cartesian grip point coordinates and rotation

7.2 Grip point

The grip point location of a generic object is simply computed with trivial geometrical operations: Figure 7.1 shows on the right the master piece characterized by its orientation and the master grip point (blue dot) rotated by an angle θ_{g1} (master grip angle) with respect to the horizontal, while on the left it shows a generic target piece, rotated by the difference angle θ with respect to the master. The target grip point (green dot) has coordinates x_{g2} and y_{g2} and it is rotated by an angle θ_{g2} (target grip angle) with respect to the horizontal.

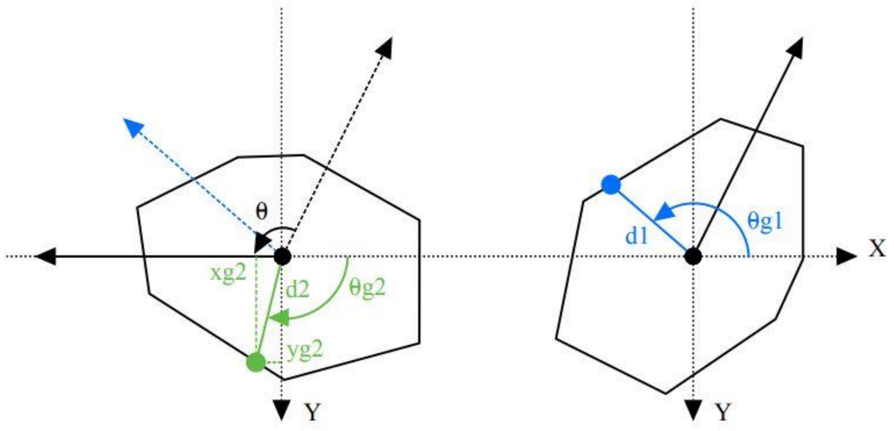


Figure 7.1: *target grip point coordinates and angle*

First, to obtain x_{g2} and y_{g2}, θ_{g2} is computed as:

$$\theta_{g2} = \theta_{g1} + \theta \tag{7.1}$$

where θ_{g1} is the master grip angle obtained from the master grip point definition in Chapter 5. The equation 6.5 is applied to θ_{g2} to be sure that the ±180° convention is achieved.

Once obtained θ_{g2}, the grip point coordinates on the target side are then computed as:

$$\begin{cases} x_{g2} = d_2 \cos\theta_{g2} \\ y_{g2} = d_2 \sin\theta_{g2} \end{cases} \tag{7.2}$$

where it is supposed $d_1 \cong d_2$. This is an important assumption: because of illumination or perspective effects that afflict the blob shape, those distances could vary of some pixels, since the blob centroid is not exactly the same for the two pieces. However, in practice, this approximation does not compromise the result in a significant way.

7.3 Canny edge detection for obstacle avoidance area

To check the presence or the absence of obstacles in the collision avoidance area, the Canny edge detection method is applied: this algorithm, largely used for this kind of applications, searches high intensity changes in an image and classifies them as edges with the assumption that the background colour is considered uniform and with a high contrast with respect to the piece. This assumption is easily realizable for this kind of applications.
Below a brief schematic explanation of how this method works, according to the documentation provided by OpenCV [53]:

1) Given an image, a 5X5 Gaussian filter is applied to remove noise, since edge detection is an operation quite susceptible to noise effects.

2) A Sobel kernel is used to compute the horizontal and vertical first derivatives of intensity, G_x and G_y, from which the edge gradient G and the direction angle γ are computed as:

$$G = \sqrt{G_x^2 + G_y^2} \tag{7.3}$$

$$\gamma = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{7.4}$$

3) Every pixel is then checked in the gradient direction to verify if it is a local minimum in its neighbourhood and, consequently, considerable as a possible edge.

4) The possible edges gradients are compared to two thresholds called minVal and maxVal: if an edge gradient is greater than maxVal it is for sure and edge, if it is lower than minVal, it is not an edge. Otherwise, if it falls between the two thresholds, it is classified edge or non-edge according to its connectivity: if it is connected to sure edges, it is an edge as well, otherwise it is a non-edge.

Figure 7.2 shows the result of this method applied, as an example, to the master image: the edges appear as contours-like white lines on a black background.

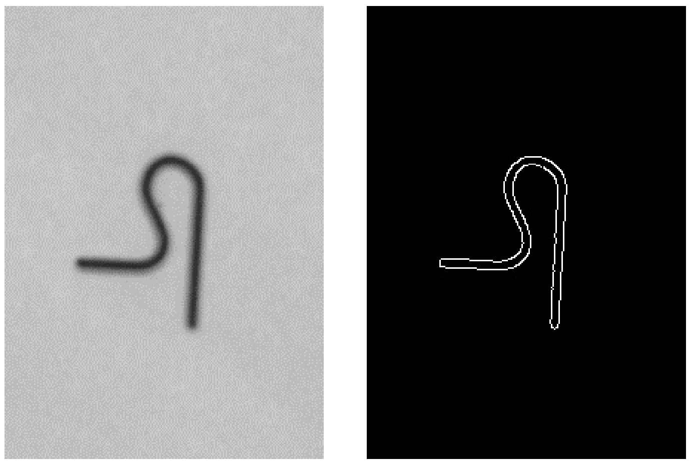


***Figure 7.2**: Canny edge method applied to the master image*

The application of this method to find obstacles in the collision avoidance area is quite simple: it is enough to apply the Canny edge method to the target piece and check the presence of edges by counting the number of white pixels: the absence of white pixels means no detected obstacles.
The tricky part is to limit the application of the Canny edge method to the collision avoidance area only.
To explain in a clearer way this procedure, it is better to adopt again a schematic explanation since some passages are non-trivial, by showing the resulting image of each step, where the inner and the outer polygons are shown just for a matter of explanation (the images used and obtained in the code do not present polygons).
To give an interesting example, it has been chosen the 17th piece referring to Figure 4.16, since it is discarded from the pickable objects list for the presence of another piece in its collision avoidance area:

1) For a generic piece, the outer and inner polygons, defined by the user in Chapter 5, are rotated of θ and each point of the outer polygon is checked to be sure that the full polygon falls inside the grip constraints.
   Figure 7.3 shows the new target image cropped in the last chapter with twice the size of grip limit. It also shows the master and the inner and outer polygons for both images.

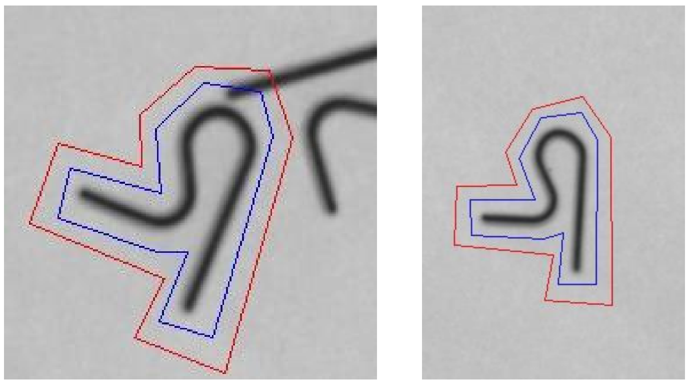


***Figure 7.3**: target image for Canny edge. It also shows the comparison between the collision avoidance area polygons on the new target (left) and on the master (right)*

2) On the target image, the Canny edge method is applied with the function cv.Canny() [54] , that receives as input the target image, minVal, and maxVal, to return the edge image shown in Figure 7.4. Here minVal and maxVal are fixed at 100 and 200.

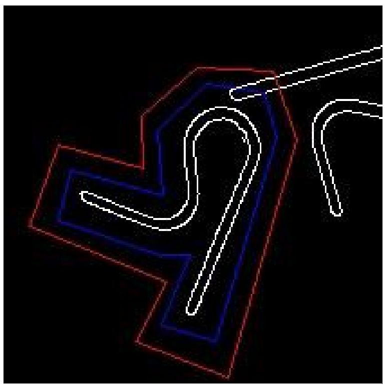


***Figure 7.4**: edge image*

3) On the edge image, the area inside the inner polygon turns black (Figure 7.5). This operation, as the next ones, are performed with the purpose to isolate the collision avoidance area from the full target image.

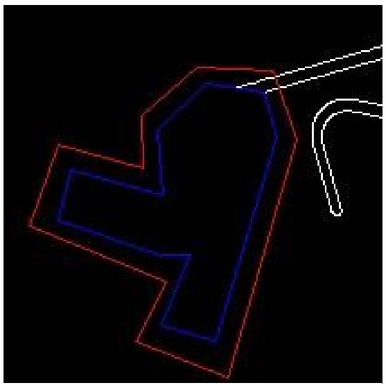


***Figure 7.5**: black inner polygon area on edge image*

4) A black stencil with same sizes of the target image (twice the grip limit) is created (Figure 7.6).

5) On the stencil, the area inside the outer polygon turns white (Figure 7.7). The fact that the images have sizes twice the grip limits guarantees that the outer polygon is drawable no matter the θ value.

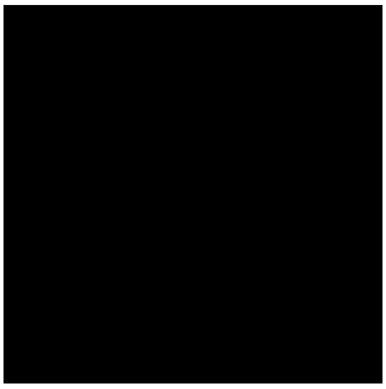


Figure 7.6*: black stencil*

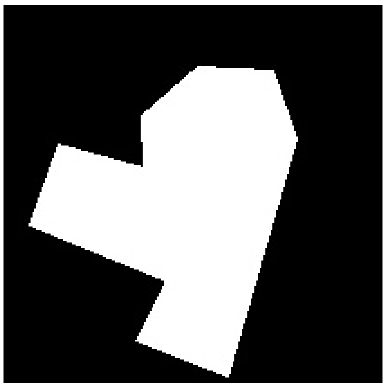


Figure 7.7*: white outer polygon area on black stencil*

6) The control mask image is created by applying the bitwise operation AND, implemented with the function cv.bitwise_and() [55]: it receives as inputs the stencil and the edge image and returns an output image where the black part of the stencil stays black, while on the white part the correspondent part of the edges image is superimposed (Figure 7.8).

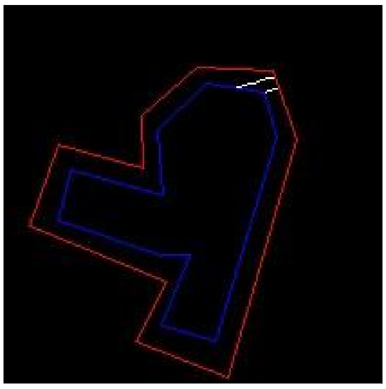


Figure 7.8*: control mask image*

7) The number of white pixels in the control mask image is counted. However, since for many reasons, as noise or strong shadows under the piece, the number of white pixels is usually non-zero even in complete absence of obstacles, a small threshold for the minimum number of white pixels is chosen, for example 15 pixels.
   In this example, in the control area there are 30 white pixels, so, the piece is not going to be considered as pickable.

The pieces that pass this last check are listed and collected as the definitive list of pickable objects.

This step implicitly introduces an important check, since it automatically excludes all the wrong-posed pieces that however present a consistent number of 'very good' match pairs. Figure 7.9 shows the results of the Canny edge detection method applied to the 9th piece, presented in the last chapter in Figure 6.8 to give an example of this kind of problem. This piece, that can be named 340-L, is the symmetric flipped version of the 340-R, and they both have quite similar features. In Figure 7.9 it can be seen how the program interprets this symmetry problem as the presence of an obstacle in the collision avoidance area.
This is a very useful property of this method, but it can be insufficient when the flipped version is much more like the master or when the collision avoidance area is defined by simpler or bigger polygons: to solve this more generic problem it is necessary to implement piece-specific controls as it is going to be discussed in Chapter 10.

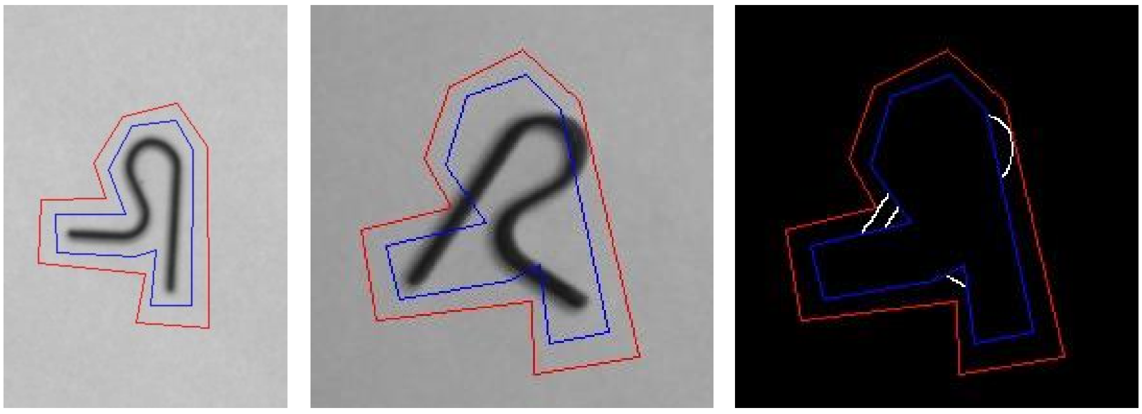


***Figure 7.9**: Canny edge on 340-L. From left to right: the master (340-R), the 9th piece (340-L), and its control mask image*

7.4 Definitive list of pickable objects

At this point, the pickable objects must be represented. Figure 7.10 shows the final result, whose legend is reported here:

- Red rectangles: minimum rectangles that identify the candidate objects blobs found in Chapter 4.

- Red dots: centroids of the blobs that respect the limits imposed in Chapter 4.

- Red arrows: result of the feature matching. Pieces without the arrow are pieces that did not pass the keypoints check. The angle in degrees with respect to the horizontal is also printed.
- Blue circles: grip points.
- Green circles: identify the pieces that also passed the collision avoidance area check.

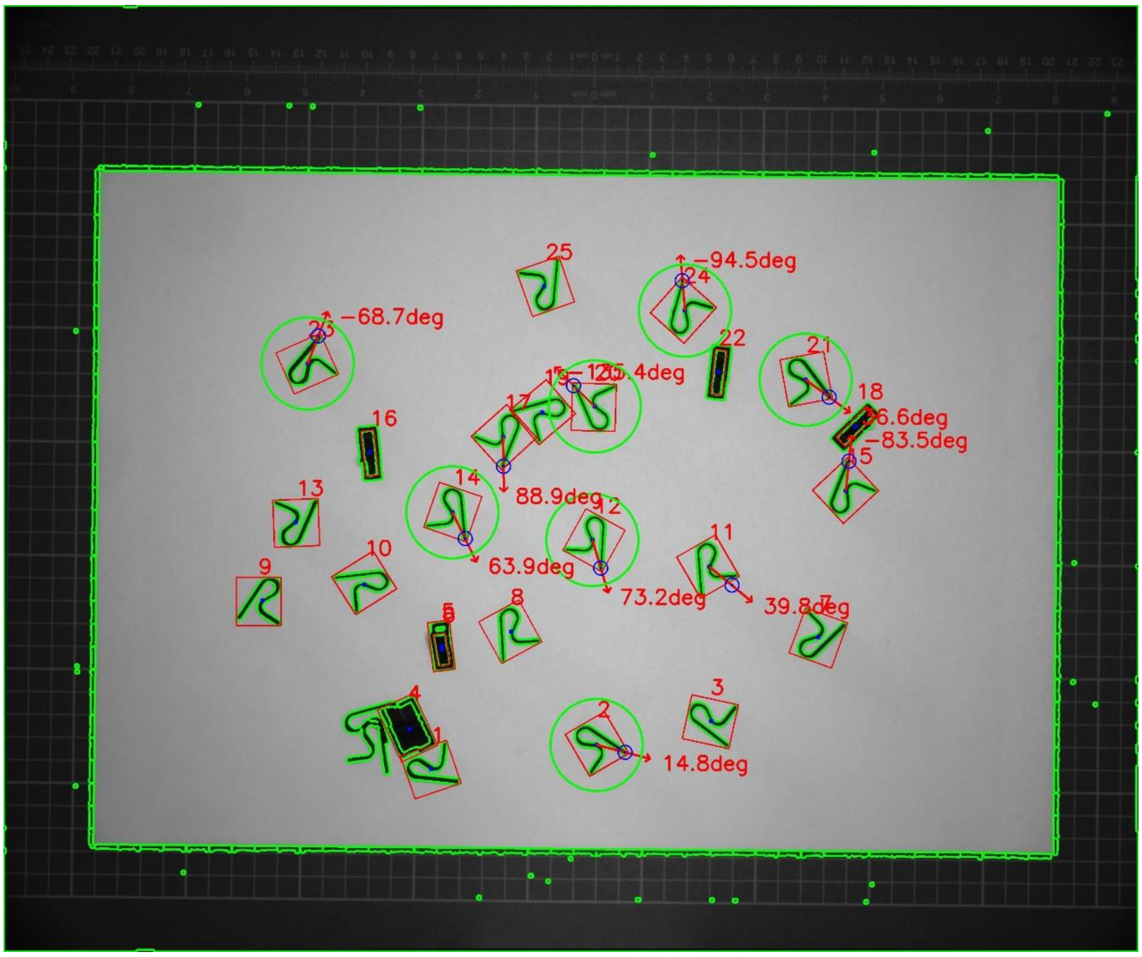


Figure 7.10: pickable objects representation on the full image

Beside the visual representation, it is also given a list of all the pickable pieces with their grip point coordinates in the full image and the difference angles with respect to the master (Table 7.1).

piece id	u [px]	v [px]	θ [deg]
2	1352	1622	-60
12	1297	1225	0
14	1003	1162	-9
20	1240	820	156
21	1797	851	-40
23	688	717	-139
24	1477	596	-166

***Table 7.1**: pickable pieces image coordinates and orientation*

7.5 Pixels-to-millimetres conversion

At this point, the last thing to do is to apply the Equations 3.9 and 3.11 to the pixel coordinates p to obtain the millimetres coordinates p_w, always considering $Z_w = 0$ and where K, R, s, and t are the parameters found in Chapter 3.

$$p_w = (KR)^{-1}(sp - Kt) \tag{3.9}$$

$$s = \frac{[(KR)^{-1}Kt]_3}{[(KR)^{-1}p]_3} \tag{3.11}$$

The result of the conversion is tabulated in Table 7.2, where only x and y are shown, since the orientation doesn't change: this happens because the transformation of θ_1 and θ_2 in the global reference frame is the same for both the angles and it is cancelled on the two subtrahends when the difference angle is obtained (Equations 6.2 and 6.3)

piece id	x [mm]	y [mm]
2	247.57	176.34
12	168.37	188.51
14	156.7	247.24
20	87.58	201.1
21	92	90
23	68.77	311.29
24	41.04	94.76

***Table 7.2**: pickable pieces world coordinates*

Chapter 8: density

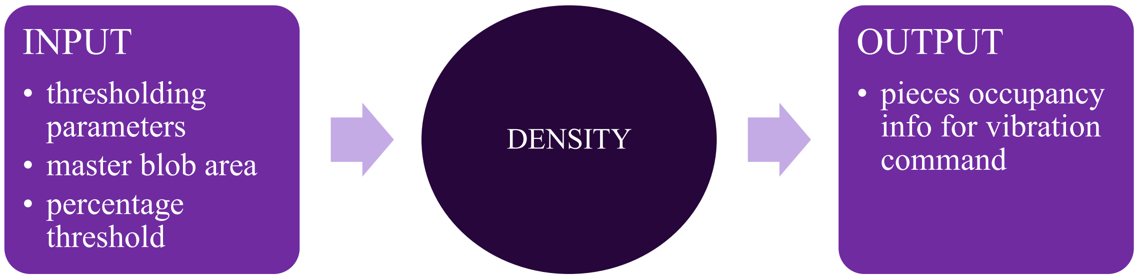
8.1 Introduction

This is the last chapter before the experimentation part that will provide the practical results of what has been analysed and described until now.
In the introduction it has been presented the Supata® machine as a robot that picks pieces from an automated platform that randomly redistributes them through a vibrating action. The peculiarity of the platform is that it can vibrate in different ways according to the distribution of the remaining pieces, thanks to its multi-motors system: for example, if at the end of the gripping phase, there are a lot of non-pickable objects of the right side of the platform, it vibrates to redistribute them in the most uniform possible way, activating the left-motor.
Thanks to the vision software described below the operations to move the pieces on the vibrating plane and to load pieces on it are completely automatic and the system adapts to the real current state.

The main goal of this last part of the code is to give to the machine an indicative information of how the pieces are distributed in the left and in the right half of the platform in percentage. According to this distribution, four possibilities are considered:

- If the left side is empty and the right side is full, the right motor activates, moving the pieces to the left.
- If the right side is empty and the left side is full, the left motor activates, moving the pieces to the right.
- If both sides are full but all the pieces are not pickable, both the motors activate to overturn or to distance the pieces.
- If both sides are empty, new pieces are loaded on the platform.

This information is achieved through well-known procedures adopted in the previous chapters: thresholding, contours detection, and white pixels counting.


INPUT
• thresholding parameters
• master blob area
• percentage threshold
DENSITY
OUTPUT
• pieces occupancy info for vibration command

8.2 Density computation

Given an image, the thresholding procedure, characterized by the same settings adopted in Chapter 4 to find the candidate object list, is applied to isolate the pieces from the background but, since the counting of white pixels results to be simpler, the thresholding output image is inverted by applying the simple bitwise operation NOT with the function cv.bitwise_not() [56] (Figure 8.1).

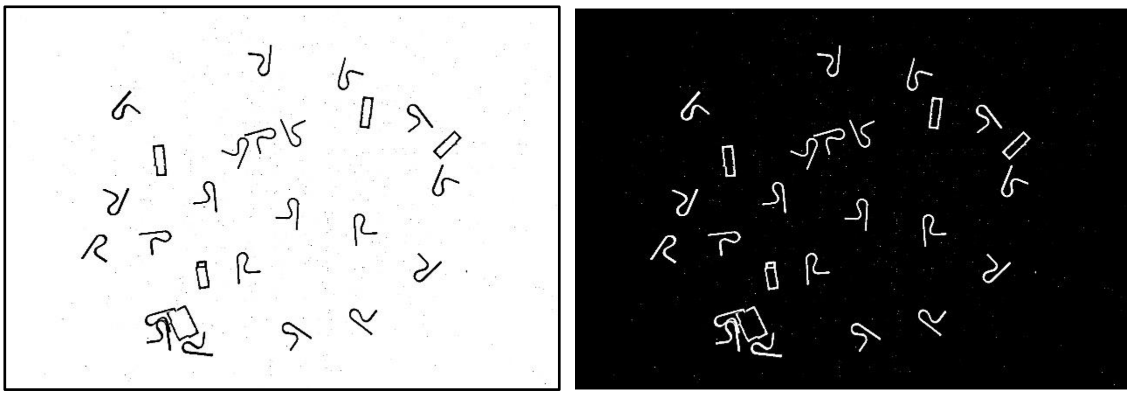


Figure 8.1: *full image thresholding for density. The result of thresholding procedure adopted in Chapter 4 (left) is inverted (right) to easily count white pixels*

Then, the closed contours are filled to obtain a more precise representation on the blobs (Figure 8.2) and the resulting image is subdivided into the half-left image and the half-right image as shown in Figure 8.3.

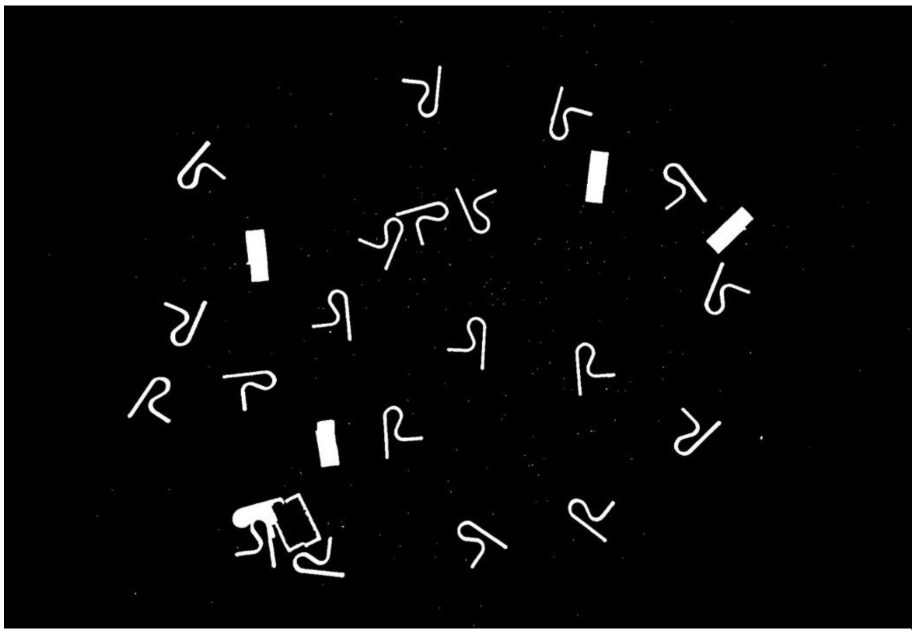


Figure 8.2: *full density image*

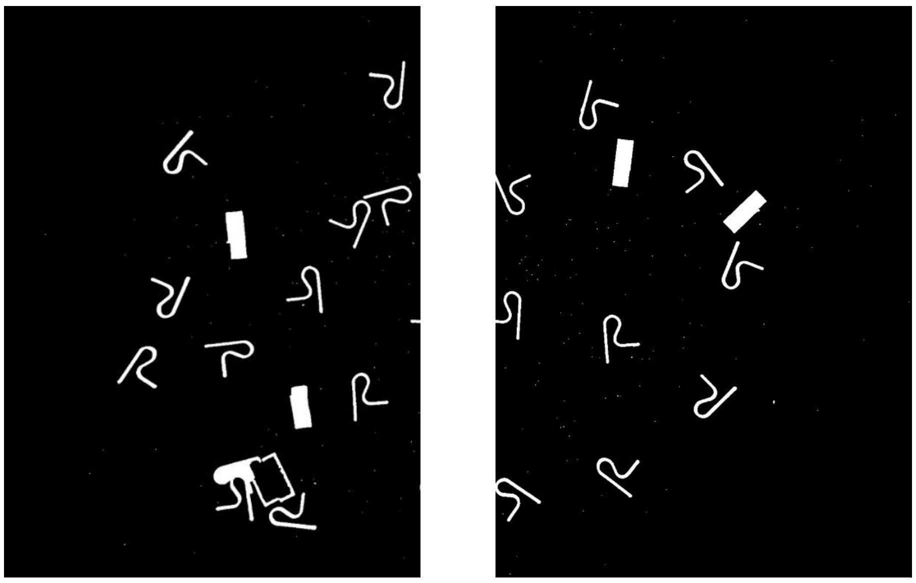


Figure 8.3: *left half and right half of the density image*

To obtain the number of pieces in the two sides, it is sufficient to separately count the number of white pixels in the left and in the right side and divide the two results by the master blob area (also obtained in Chapter 4). Then, to compute the pieces percentage, it is enough to multiply the two values by 100 and divide them by their sum. For example, the results for Figure 8.3 are:

- White pixels left = 46498
- White pixels right = 33038
- Master blob area = 2582.5

- Number of computed pieces (left, right) = (18, 13)
- Number of counted pieces (left, right) = (16, 11)
- Percentage of computed pieces (left, right) = (58%, 42%)
- Percentage of counted pieces (left, right) = (59%, 41%)

By comparing the computed and the counted values, it is possible to see that the differences are quite small and largely acceptable, since the result must just provide an idea of the pieces distribution.
However, since the area of a blob can vary a lot depending on its corresponding piece pose, it may happen that a blob that appears much greater that the master blob is interpreted by the program as a collection of pieces, while it corresponds to just one completely wrong-posed piece. To improve the approximation correctness in this cases, the user can set a corrective multiplicative parameter (1 by default) applied to the number of computed pieces that empirically increases or reduces the final percentages.

At this point, once the two percentages have been obtained, the user must set two thresholds, T_A and T_B, that defines the vibrating mode ranges of the platform. If N_1 is the number of computed pieces in the left side, N_2 is the number of computed pieces in the right side, and N_p is the total number of pickable pieces, the four possibilities are:

- $N_1 < T_B$ and $N_2 > T_A$: move the pieces to the left.
- $N_1 > T_A$ and $N_2 < T_B$: move the pieces to the right.
- $N_1 > T_A$, $N_2 > T_A$ and $N_p = 0$: overturn/distance the pieces
- $N_1 < T_A$ and $N_2 < T_A$: load new pieces.

Chapter 9: tests

9.1 Tests presentation

This chapter presents and describes the results of the open-source OpenCV-based code discussed until now. To understand their quality, those results are compared with the ones obtained with the Cognex industrial vision software currently employed on Supata® machine that will be considered as a baseline.
To obtain a significant number of data, it has been decided to consider four pieces with different characteristics, tabulated in Table 9.1, whose masters are represented in Figure 9.1. For convenience, the pieces will be named from now on as 340-R (right), diapason188, filo033, and fioregrigia:

- 340-R: black piece on white background, classified as matte since it poorly reflects light. Its R-like shape has no axes of symmetry, and its thinness leads to a low number of detected keypoints with respect to the other pieces: for this reason, it has been chosen as starting piece for the empirical choice of the ORB/SIFT functions parameters.

- diapason188: silvery-shiny piece, very reflective to the light source. Since it appears white, a green background has been adopted. Differently from 340-R, it presents a larger area and more interesting features such as the central hole or the terminal groove. It also does not present any symmetries.

- filo033: grey piece, white background, low reflectivity. Even if it can be classified as thin, the number of interesting features is quite high because of its complex shape. Unlike the previous pieces, it presents one axis of symmetry that could lead to ambiguity in the grip point location: this problem is easily solved by excluding the two ends from the keypoint area.

- fioregrigia: grey piece, black background. By comparing Figure 9.5 with Figure 9.3, 9.2, and 9.4, it is possible to see that this piece has significantly bigger dimensions compared to the other pieces. Even if it is not shiny, it presents high illumination gradients, due to the complex shape of the piece. Like filo033, it presents one axis of symmetry and, consequently, problems for the grip point location: to solve this problem it is possible to consider in the keypoint area only the central part that appears as a white square-like shape. However, this piece is affected by another issue, largely discussed in the next chapter, that leads to the identification of wrong-posed pieces as pickable: this issue cannot be easily solved by the techniques discussed until now.

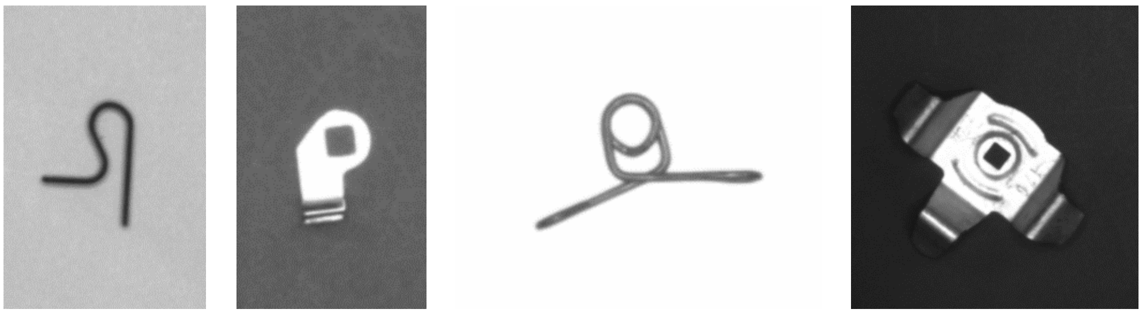


Figure 9.1: *master images. From left to right: 340-R, diapason188, filo033, fioregrigia*

Table 9.2 tabulates the camera exposure time in milliseconds and the gain chosen to obtain the images. These parameters are of major importance since they define the illumination conditions and the contrast characterizing the full images.

	340-R	diapason188	filo033	fioregrigia
shape	thin	normal	thin	wide
symmetries	0	0	1	1
piece colour	black	grey	grey	grey
piece texture	matte	shiny	matte	matte
background colour	white	green	white	black

Table 9.1: *pieces features*

	340-R	diapason188	filo033	fioregrigia
exposure time [ms]	90	100	120	120
gain	70	70	70	70

Table 9.2: *setup features*

Then, the thresholding characteristics defining the candidate object detection (Table 9.3) and the ORB and SIFT parameters (Table 9.4 and 9.5) for the feature matching phase are tabulated for every piece. All the tabulated values refer to the OpenCV functions presented in Chapter 4 for the thresholding and in Chapter 6 for the feature matching.

	340-R	diapason188	filo033	fioregrigia
thresholding	/	111	212	124
erosion kernel	/	0	0	0
adaptive parameter	21	/	/	/
inverted/normal	normal	inverted	normal	inverted
standard/adaptive	adaptive	standard	standard	standard

Table 9.3*: thresholding features*

	340-R	diapason188	filo033	fioregrigia
nfeatures	3000	3000	3000	3000
nOctaveLayers	5	5	5	5
contrastThreshold	0.003	0.01	0.003	0.003
edgeTreshold	60	60	60	100
sigma	1.6	1.6	1.6	1.6
enable_precision_upscale	True	True	True	True

Table 9.4*: SIFT parameters*

	340-R	diapason188	filo033	fioregrigia
nfeatures	500	500	500	500
scaleFactor	1.2	1.2	1.2	1.2
nlevels	10	30	10	10
edgeTreshold	5	10	50	30
firstLevel	0	0	0	0
WTA_K	2	2	2	2
scoreType	0	0	0	0
patchSize	31	31	31	31
fastThreshold	5	5	5	5

Table 9.5*: ORB parameters*

Considering all those settings, it has been decided to acquire three images for each piece so that the analysis results can be consistent.
Figure 9.2, 9.3, 9.4, and 9.5 show the four triplets (the masters are taken from the first image of each triplet).

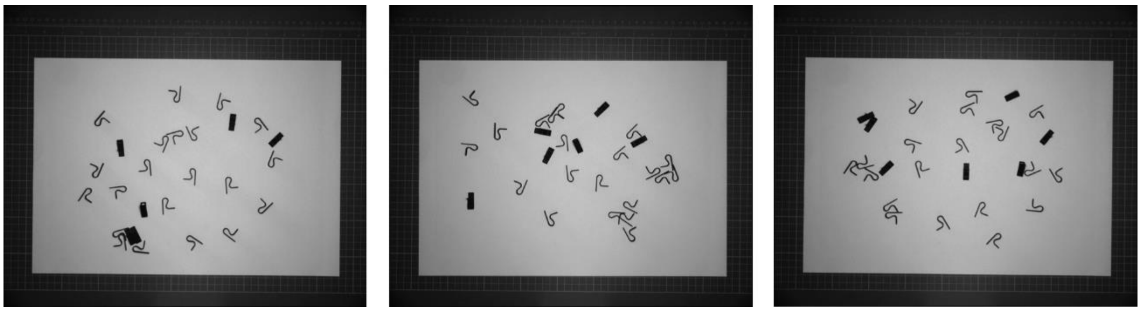


Figure 9.2*: 340-R test images*

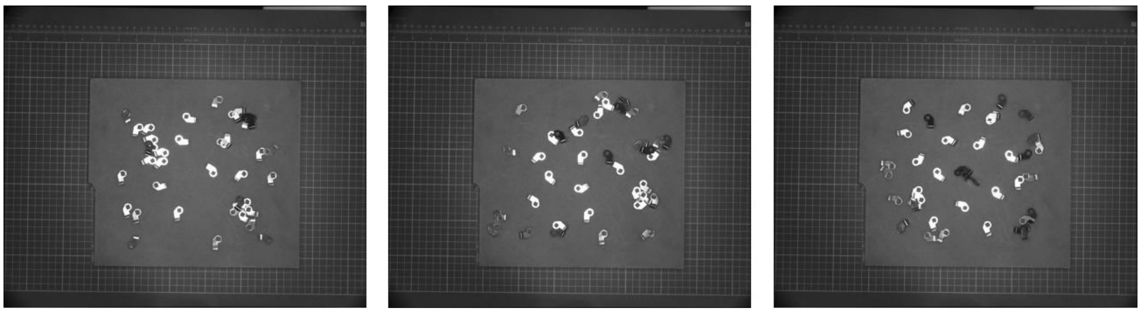


Figure 9.3*: diapason188 test images*

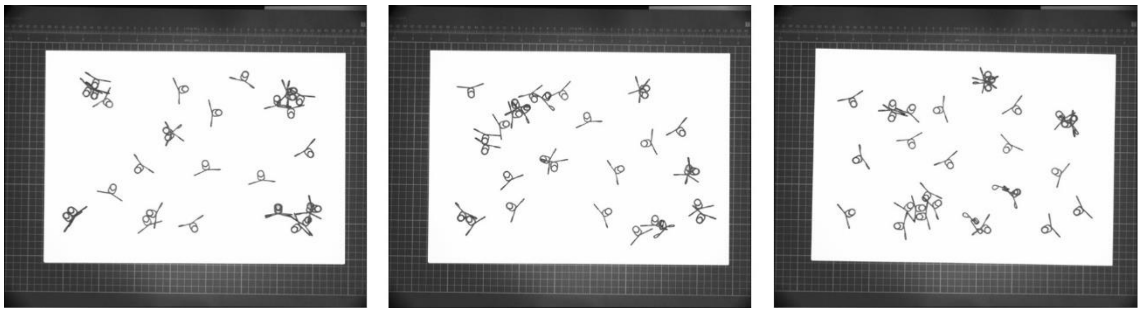


Figure 9.4*: filo033 test images*

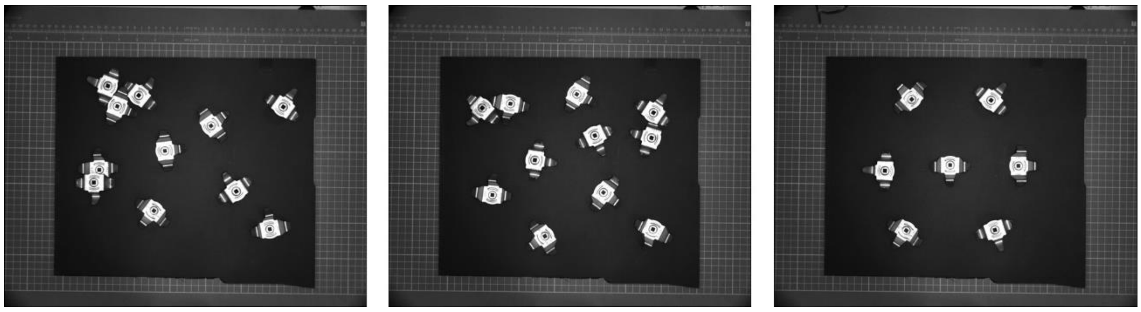


Figure 9.5*: fioregrigia test images*

9.2 Confrontation results

This paragraph shows the results obtained for each piece and the first considerations to establish which algorithm combinations works better between ORB and SIFT with Brute-Force, FLANN, and FLANN plus homography. However, since the procedure is the same every time, for every piece, and for every algorithm combination, the complete passages are shown only for the first image of the first piece (340-R), while for the other pieces only the table with the results shall be reported.

First, the Cognex results are obtained and tabulated (Table 9.6). Then, for each piece it is calculated the difference between the x and y coordinates in pixels, always indicated as u and v since they refer to the image reference system, and the rotation θ in degrees. Then, the code results are obtained using the six matching combinations. In Table 9.7, corresponding to ORB+BF, are reported the identity numbers of the piece referring to Figure 9.6, the coordinates u, v, and θ, and the modules of their difference with respect to Cognex.

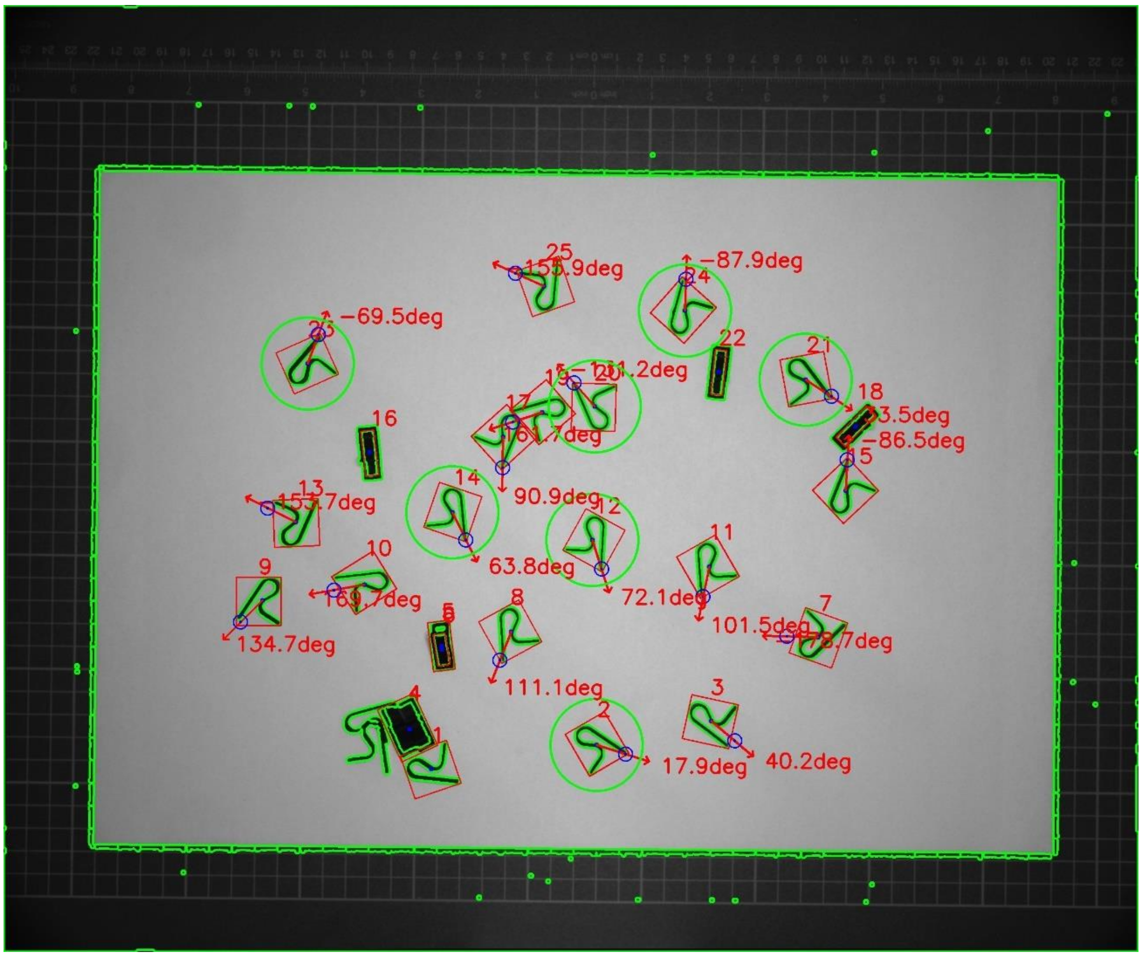


***Figure 9.6**: 340-R, 1st image with ORB+BF*

piece id	u	v	θ
2	1348	1623	-59
12	1297	1222	0
14	1001	1158	-9
20	1240	826	153
21	1792	849	-40
23	684	722	-142
24	1474	601	-168

***Table 9.6**: Cognex results for 340-R, 1st image*

To distinguish 'good' and 'bad' pieces, it has been decided that every piece with |Δθ| > 5° is considered 'bad'. No checks on |Δu| and |Δv| are performed because the grip point is manually imposed by the user in both software (this and Cognex) so, the presence of a difference of some pixels that has nothing to do with the code is highly probable: for this reason, |Δu| and |Δv| can only give a qualitative idea of the cartesian coordinates differences.

piece id	u	v	θ	|Δu|	|Δv|	|Δθ|
2	1351	1626	-57	3	3	2
12	1299	1225	-2	2	3	2
14	1004	1161	-10	3	3	1
20	1236	823	152	4	3	1
21	1797	850	-40	5	1	0
23	683	715	-143	1	7	1
24	1482	596	-161	8	5	7

***Table 9.7**: ORB+BF results for 340-R, 1st image. The row underlined in blue corresponds to the master, while the row underlined in orange corresponds to a 'bad piece', since the angle difference is greater than 5° (red cell).*

To verify if one algorithm is better than the others, this procedure is performed for all the other five combinations. Then, for each one the mean of the differences is computed by considering only the 'good pieces' as:

$$\begin{cases} \mu_u = \overline{|\Delta u|} \\ \mu_v = \overline{|\Delta v|} \\ \mu_\theta = \overline{|\Delta \theta|} \end{cases} \tag{9.1}$$

Figure 9.7 shows the result of the μ analysis for all the pieces and for all the algorithm combinations.

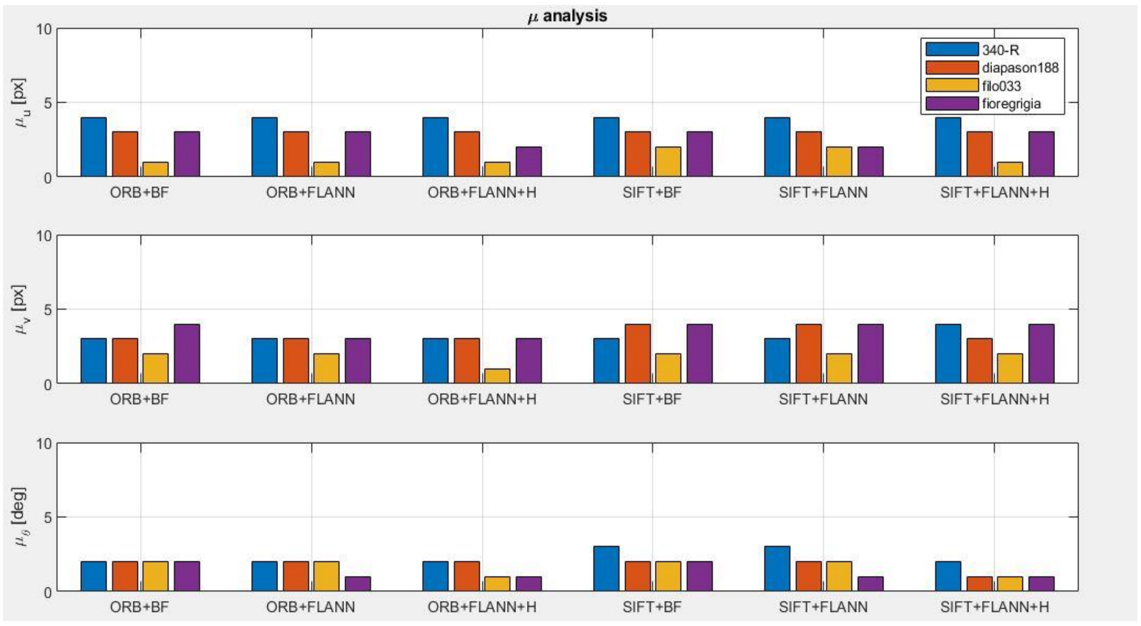


***Figure 9.7**: μ analysis histogram*

From this first comparison, it is possible to see that the three μ values are quite similar for all the algorithm combinations, so, it is mandatory to proceed with the bad pieces analysis to decide which one is better.
However, before showing the results concerning the bad pieces analysis, it is necessary to explain how they are classified.

- BP (bad pieces): all the pieces that for any reason cannot be considered. They include OTP, WP, and MP.
- OTP (out of tolerance pieces): pieces for which $5° < |\Delta\theta| \leq 10°$.
- WP (wrong pieces): pieces for which $|\Delta\theta| > 10°$.
- MP (missed pieces): pieces not recognized as pickable.

Table 9.8, 9.9, 9.10, and 9.11 present BP, for each algorithm and for each piece, as absolute number of bad pieces and as percentage referred to the total number of pieces, while OTP, WP, and MP are expressed in percentages referred to BP. Their absolute value is shown in Figure 9.8.

	μ_u	μ_v	μ_θ	BP		OTP	WP	MP
ORB+BF	4	3	2	2	11%	100%	0%	0%
ORB+FLANN	4	3	2	1	6%	100%	0%	0%
ORB+FLANN+H	4	3	2	0	0%	/	/	/
SIFT+BF	4	3	3	5	28%	40%	60%	0%
SIFT+FLANN	4	3	3	6	33%	50%	50%	0%
SIFT+FLANN+H	4	4	2	2	11%	50%	50%	0%

Table 9.8*: 340-R results*

	μ_u	μ_v	μ_θ	BP		OTP	WP	MP
ORB+BF	3	3	2	4	15%	50%	25%	25%
ORB+FLANN	3	3	2	3	12%	0%	100%	0%
ORB+FLANN+H	3	3	2	4	15%	75%	0%	25%
SIFT+BF	3	4	2	4	15%	25%	25%	50%
SIFT+FLANN	3	4	2	4	15%	25%	25%	50%
SIFT+FLANN+H	3	3	1	5	19%	40%	0%	60%

Table 9.9*: diapason188 results*

	μ_u	μ_v	μ_θ	BP		OTP	WP	MP
ORB+BF	1	2	2	1	4%	100%	0%	0%
ORB+FLANN	1	2	2	0	0%	/	/	/
ORB+FLANN+H	1	1	1	0	0%	/	/	/
SIFT+BF	2	2	2	7	25%	86%	14%	0%
SIFT+FLANN	2	2	2	7	25%	86%	14%	0%
SIFT+FLANN+H	1	2	1	1	4%	100%	0%	0%

Table 9.10*: filo033 results*

	μ_u	μ_v	μ_θ	BP		OTP	WP	MP
ORB+BF	3	4	2	3	23%	0%	100%	0%
ORB+FLANN	3	3	1	6	46%	50%	50%	0%
ORB+FLANN+H	2	3	1	0	0%	/	/	/
SIFT+BF	3	4	2	4	31%	50%	50%	0%
SIFT+FLANN	2	4	1	5	38%	40%	60%	0%
SIFT+FLANN+H	3	4	1	1	8%	0%	100%	0%

Table 9.11*: fioregrigia results*

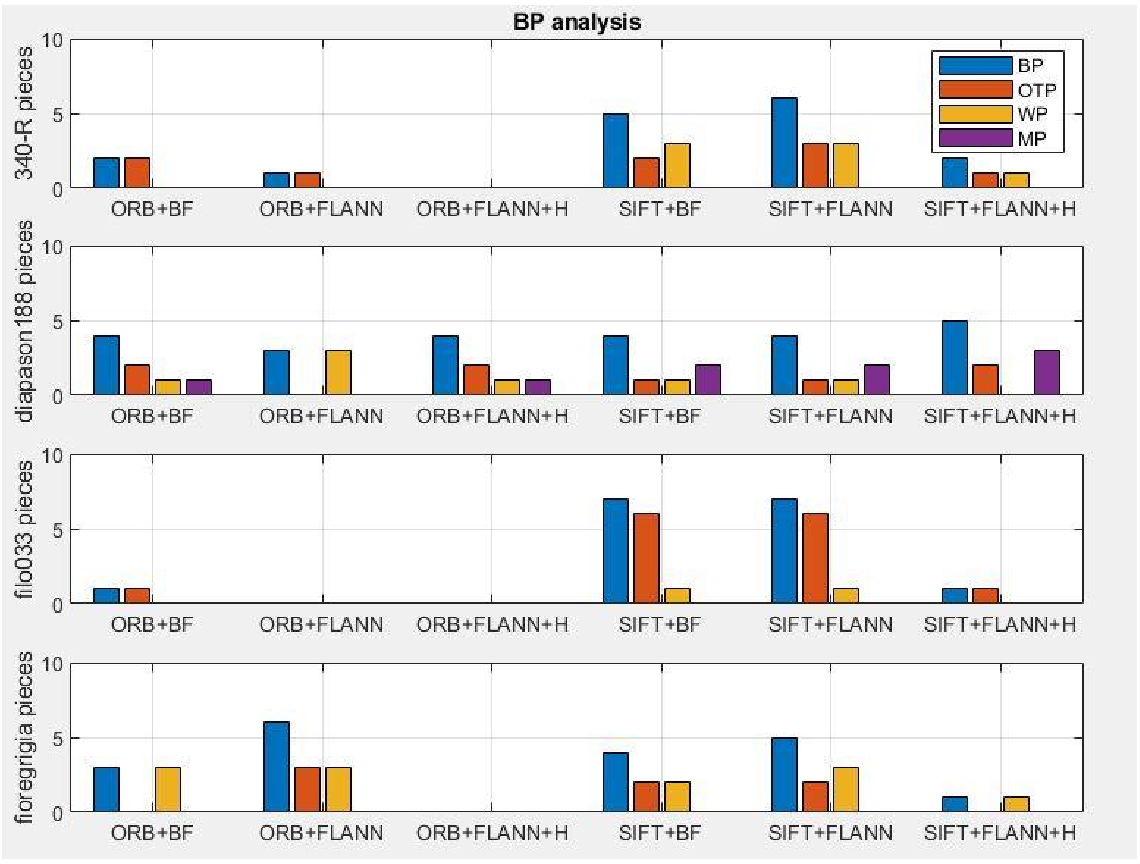


***Figure 9.8**: BP analysis histogram*

It is clear, according to this last histogram, that the best algorithms are the two that adopt the FLANN matcher + homography combination: ORB+FLANN+H and SIFT+FLANN+H.
ORB+FLANN+H presents the lowest number of BP, all included in the diapason188 case that is the most sensitive piece to the illumination conditions, since it is the only shiny piece. This highlights how the illumination conditions are important for the matching results since the matching function parameters depend on those conditions. It is reasonable to think that this algorithm could be able to correctly find all the pieces by finding the correct external parameters settings, for example, reducing the exposure time (reduce luminosity), changing the background colour and, consequently, find the correct ORB parameters combination.

9.3 Computational time

To choose an algorithm instead of another, the accuracy is not enough: it is also fundamental to consider the time each one takes to perform the feature matching. For each image and for each algorithm combination the computational time in seconds has been obtained: again, Table 9.12 shows the procedure just for 340-R, but it has been repeated in the same way for all the other pieces.

	image 1	image 2	image 3
ORB+BF	0,799	0,774	0,758
ORB+FLANN	0,793	0,794	0,775
ORB+FLANN+H	0,311	0,322	0,252
SIFT+BF	1,345	1,239	1,336
SIFT+FLANN	2,739	2,477	2,616
SIFT+FLANN+H	2,953	2,791	3,069

***Table 9.12**: 340-R time analysis*

Once all data are collected the mean μ_t and the standard deviation σ_t over all the twelve images have been computed. The results are shown in Table 9.13 and in Figure 9.9.

	μ_t [s]	σ_t
ORB+BF	0,737	0,034
ORB+FLANN	0,756	0,048
ORB+FLANN+H	0,223	0,052
SIFT+BF	0,728	0,372
SIFT+FLANN	1,289	0,859
SIFT+FLANN+H	1,406	0,975

***Table 9.13**: global time analysis*

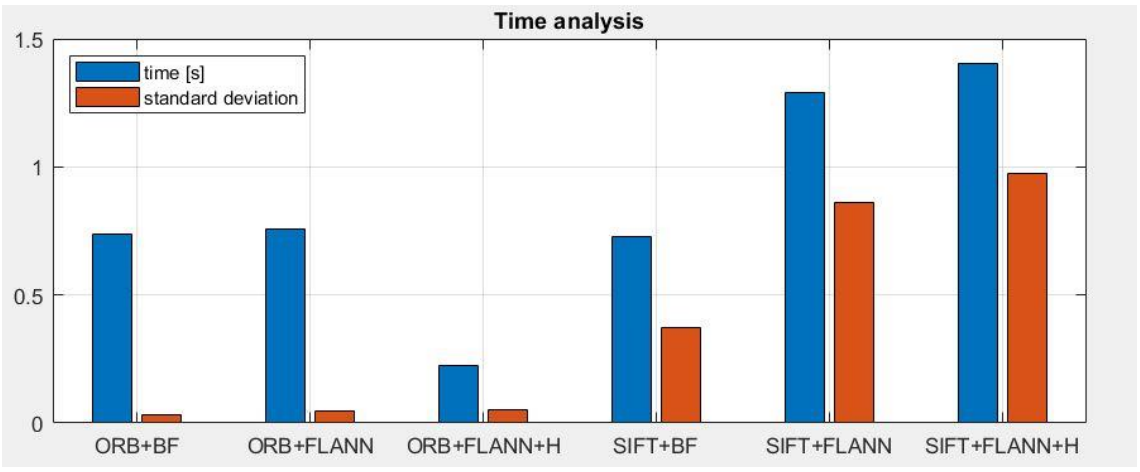


Figure 9.9*: computational time histogram*

From this analysis it is clear how the ORB+FLANN+H algorithm is the best without any possible comparison with SIFT+FLANN+H, which turns out to be the worst one, with a difference of more than a second. However, this is not the only pro: all the ORB algorithms have a standard deviation σ_t much smaller than the SIFT one: this means that the SIFT computational time depends a lot on the image characteristics, while the ORB computational time is not only lower, but it is also less affected by image content and more robust to noise and variations.

As a result, it is clear how the best choice is the ORB+FLANN+H algorithm.

Chapter 10: conclusions

The goal of this thesis project was to develop a free-of-license, open-source industrial vision application for robot guidance, able to detect well-posed pieces and then provide the coordinates of the grip point and the orientation of the pieces to a manipulator, no matter the dimensions, the material, or the pieces shape.
The discussion started with the presentation of standard vision operations such as camera calibration and pose estimation and it proceeded with the description of the main program, from the initial settings to the achievement of the final output. Eventually, Chapter 9 has described how the ORB+FLANN+H algorithm combination has the best performance in terms of precision, number of well-posed pieces detected, and computational time, revealing that the combination mentioned above might be a promising starting point for future improvements.

In fact, given the results of the tests, it can be said that this software is a valid base for a new series of improvements aiming to reduce errors and to increase the performance quality of each step in the entire process, such as:

- Improvement of the calibration capabilities: currently, the distortion model does not consider the perspective effect both between camera plane and object plane and between camera lens and camera sensor. This would allow to use worse hardware by compensating with the software and would generalize for different camera mount positions, also improving the pose estimation precision.
  .
- Improvement of the pose estimation precision also by adopting high-precision instrumentation to reduce as possible the angular error on the camera and, consequently, the pose estimation error. Moreover, it can be made more user-friendly, letting the user to impose the origin frame location and the axis orientation, independently by the ChArUco standards.

- Improvement of the feature matching precision, focusing on the ORB+FLANN+H algorithm. Referring to the tests, this critical aspect can be expressed as the reduction of |Δu|, |Δv|, and |Δθ|, and can be achieved by increasing the number of detected keypoints and well-matched pairs by finding the best settings for the ORB+FLANN+H parameters.

- Introduction of a specific function based on image resolution, illumination conditions, contrast, piece characteristics, and background colour that automatically sets the optimal ORB+FLANN+H parameters. This is probably the most time-demanding aspect, since it requires a great number of experiments with different cases.

- Reduction of the computational time by optimizing the code.

- Introduction of the possibility of implementing specific control functions for problematic pieces. A very effective example to explain this problem is the fioregrigia, where some pieces that are turned upside down are recognized as pickable, as it can be seen in Figure 10.1.
  This happens because the 2D projection of a well-posed fioregrigia is almost identical to the projection of a flipped piece and the program does not distinguish between the features of a well-posed piece and its flipped version. Even for a human operator this would be a hard task by simply looking at the picture.
  To solve this ambiguity, it is then necessary to implement a piece-specific control code to check for small, known details.

- Management of several masters at the same time. An example of this implementation can be the 340 pieces: the 340-R has been considered during the whole project but, with this implementation, the 340-L can be considered at the same time as well.

- Implementation of the AI to improve matching precision and learning capability of the program so that it can adapt to every type of piece.

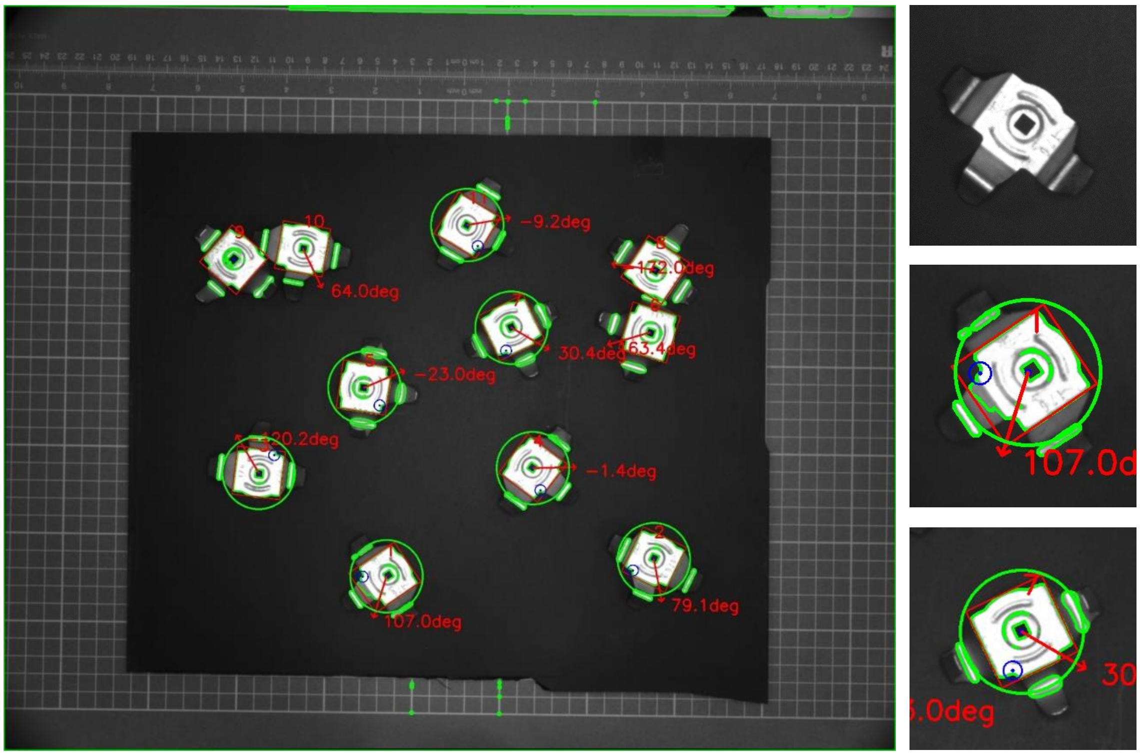


Figure 10.1: *fioregrigia problem. The full image (left) corresponds to the resulting image of the ORB+FLANN+H algorithm applied to the 2nd fioregria image. On the right, from top to bottom: the master, a well-posed piece, and a wrong-posed piece. Although the wrong-posed piece is flipped with respect to the master, it is recognized as pickable since the features in the keypoint area (white central area) are very similar to the well-posed piece ones*

Bibliography

1.1 - Intel, *What is machine vision*, (https://www.intel.com/content/www/us/en/manufacturing/what-is-machine-vision.html)

1.2 [1] EPF, *Supata®*, (https://www.epf.it/en/supata/)

1.3 - OpenCV, *About*, (https://opencv.org/about/)
- OpenCV, *Image processing on OpenCV*, (https://docs.opencv.org/4.x/d2/d96/tutorial_py_table_of_contents_imgproc.html)

[2] OpenCV, *Media Kit*, (https://opencv.org/resources/media-kit/)

2.2 - Automate, *GigE Vision Standard*, (https://www.automate.org/a3-content/vision-standards-gige-vision)

- Wikipedia, *GigE Vision*, (https://en.wikipedia.org/wiki/GigE_Vision)

[3] Wikipedia, *GigE Vision Logo*, (https://de.wikipedia.org/wiki/GigE_Vision#/media/Datei:GigE_Vision_Logo.svg)

[4] Cognex CAM-CIC-5000-20-CG datasheet

[5] Basler, *Basler ace acA2500-14gc*, (https://www.baslerweb.com/en/products/cameras/area-scan-cameras/ace/aca2500-14gc/#specs)

[6] Edmund Optics, *16mm UC Series Fixed Focal Length Lens*, (https://www.edmundoptics.eu/p/16mm-uc-series-fixed-focal-length-lens/2970/)

- GitHub, *Harvesters*, (https://github.com/genicam/harvesters#about-harvester)

[7] EMVA, *GenICam introduction*, (https://www.emva.org/standards-technology/genicam/introduction-new/)

2.3 - Wikipedia, *Pinhole camera model*, (https://en.wikipedia.org/wiki/Pinhole_camera_model)

- OpenCV, *Camera Calibration*, (https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html)

[8] OpenCV, *Camera Calibration and 3D Reconstruction*, (https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html)

2.4 - OpenCV, *Detection of ArUco Boards*, (https://docs.opencv.org/4.x/db/da9/tutorial_aruco_board_detection.html)

[9] OpenCV, *Detection of ChArUco Boards*, (https://docs.opencv.org/4.x/df/d4a/tutorial_charuco_detection.html)

[10] GitHub, *OpenCVMarkerPrinter*, 2019, Josh Chien,
[11] (https://github.com/dogod621/OpenCVMarkerPrinter)

[12] OpenCV, *cv::aruco::CharucoBoard Class Reference*, (https://docs.opencv.org/3.4/d0/d3c/classcv_1_1aruco_1_1CharucoBoard.html)

- OpenCV, *Detection of ArUco Markers*, (https://docs.opencv.org/3.4/d5/dae/tutorial_aruco_detection.html)

- OpenCV, *Camera Calibration*, (https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html)

[13] OpenCV, *Aruco markers, module functionality was moved to objected module*, (https://docs.opencv.org/4.x/d9/d6a/group__aruco.html#ga3bc50d61fe4db7bce8d26d56b5a6428a)

[14] OpenCV, *ArUco Marker Detection*,
[15] (https://docs.opencv.org/3.4/d9/d6a/group__aruco.html)

[16] OpenCV, *Camera Calibration and 3D Reconstruction*,
[17] (https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html)

3.1 - OpenCV, *Pose Estimation*, (https://docs.opencv.org/3.4/d7/d53/tutorial_py_pose.html)

3.2 [18] OpenCV, *Camera Calibration and 3D Reconstruction*, (https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html)

- Stackoverflow, *Computing x,y coordinate (3D) from image point*, (https://stackoverflow.com/questions/12299870/computing-x-y-coordinate-3d-from-image-point)

3.3 [19] OpenCV, *ArUco Marker Detection*, (https://docs.opencv.org/3.4/d9/d6a/group__aruco.html#gab098ca624829bcbf7d9ebb8479887c3a)

[20] OpenCV, *Camera Calibration and 3D Reconstruction*, (https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#gab3ab7bb2bdfe7d5d9745bb92d13f9564)

- OpenCV, *Camera Calibration*, (https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html)

[21] OpenCV, *Camera Calibration and 3D Reconstruction*, (https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga1019495a2c8d1743ed5cc23fa0daff8c)

[22] OpenCV, *ArUco Marker Detection*, (https://docs.opencv.org/3.4/d9/d6a/group__aruco.html#ga0c158c55c50df8354930927d819f7e9d)

[23] OpenCV, *Camera Calibration and 3D Reconstruction*, (https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga61585db663d9da06b68e70cfbf6a1eac)

[24] MathWorks, *rotm2eul()*, (https://it.mathworks.com/help/robotics/ref/rotm2eul.html)

4.4 - Matplotlib, (https://matplotlib.org/)

4.5 [25] OpenCV, *Media Kit*, (https://opencv.org/resources/media-kit/)

- Wikipedia, *Thresholding (image processing)*, (https://en.wikipedia.org/wiki/Thresholding_(image_processing))

[26] [27] OpenCV, *Image Thresholding*, (https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html)

[28] [29] OpenCV, *Miscellaneous Image Transformations*, (https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html)

[30] OpenCV, *Eroding and Dilating*, (https://docs.opencv.org/3.4/db/df6/tutorial_erosion_dilatation.html)

[31] OpenCV, *More Morphology Transformations*, (https://docs.opencv.org/3.4/d3/dbe/tutorial_opening_closing_hats.html)

4.6 - OpenCV, *Contours: Getting Started*, (https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html)

[32] [33] [34] OpenCV, *Structural Analysis and Shape Descriptors*, (https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html)

6.2 [35] OpenCV, *Understanding Features*, (https://docs.opencv.org/4.x/df/d54/tutorial_py_features_meaning.html)

6.3 [36] *Distinctive Image Features from Scale-Invariant Keypoints*, January 5th, 2004, David G. Lowe

- Medium, *Introduction to SIFT (Scale Invariant Feature Transform)*, March 16th, 2019, Deepanshu Tyagi, (https://medium.com/data-breach/introduction-to-sift-scale-invariant-feature-transform-65d7f3a72d40)

- Wikipedia, *Scale-Invariant feature transform*, (https://en.wikipedia.org/wiki/Scale-invariant_feature_transform)

[37] [38] OpenCV, *Introduction to SIFT (Scale-Invariant Feature Transform)*, (https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html)

[39] OpenCV, *Feature Matching with FLANN*, (https://docs.opencv.org/3.4/d5/d6f/tutorial_feature_flann_matcher.html)

6.4 [40] *ORB: An efficient alternative to SIFT or SURF*, 2011, Ethan Rublee, Vincent Rabaud, Kurt Konolige, Gary R. Bradski

- OpenCV, *ORB (Orientaed Fast and Rotated BRIEF)*, (https://docs.opencv.org/3.4/d1/d89/tutorial_py_orb.html)

- Medium, *Introduction to ORB (Orientaed Fast and Rotated BRIEF)*, January 1st, 2019, Deepanshu Tyagi, (https://medium.com/data-breach/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf)

6.5

- OpenCV, *Feature Matching*, (https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html)

- OpenCV, *Feature Matching + Homography to find Objects*, (https://docs.opencv.org/4.x/d1/de0/tutorial_py_feature_homography.html)

6.6 [41] OpenCV, *cv::SIFT Class Reference*, (https://docs.opencv.org/4.x/d7/d60/classcv_1_1SIFT.html)

[42] OpenCV, *cv::ORB Class Reference*, (https://docs.opencv.org/4.8.0/db/d95/classcv_1_1ORB.html)

[43] [45] OpenCV, *cv::Feature2D Class Reference*, (https://docs.opencv.org/4.x/d0/d13/classcv_1_1Feature2D.html)

[44] [46] OpenCV, *cv::KeyPoint Class Reference*, (https://docs.opencv.org/4.x/d2/d29/classcv_1_1KeyPoint.html)

[47] OpenCV, *cv::BFMatcher Class Reference*, (https://docs.opencv.org/4.7.0/d3/da1/classcv_1_1BFMatcher.html)

[48] OpenCV, *cv::FlannBasedMatcher Class Reference*, (https://docs.opencv.org/3.4/dc/de2/classcv_1_1FlannBasedMatcher.html)

- OpenCV, *Feature Matching*, (https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html)

[49] OpenCV, *cv::DescriptorMatcher Class Reference*,
[50] (https://docs.opencv.org/4.7.0/db/d39/classcv_1_1DescriptorMatcher.html)

[51] OpenCV, *Camera Calibration and 3D Reconstruction*, (https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga4abc2ece9fab9398f2e560d53c8c9780)

[52] OpenCV, *Drawing Function of Keypoints and Matches*, (https://docs.opencv.org/4.x/d4/d5d/group__features2d__draw.html)

- INTELLIGENZA ARTIFICIALE ITALIA, *Come rimuovere e gestire i valori anomali con python nel machine learning*, Team I.A. Italia, (https://www.intelligenzaartificialeitalia.net/post/come-rimuovere-e-gestire-i-valori-anomali-con-python-nel-machine-learning)

7.3 [53] OpenCV, *Canny Edge Detection*, (https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html)

[54] OpenCV, *Feature Detection*, (https://docs.opencv.org/4.x/dd/d1a/group__imgproc__feature.html#ga04723e007ed888ddf11d9ba04e2232de)

[55] OpenCV, *Operations on arrays*, (https://docs.opencv.org/3.4/d2/de8/group__core__array.html#ga60b4d04b251ba5eb1392c34425497e14)

8.2 [56] OpenCV, *Operations on arrays,* (https://docs.opencv.org/3.4/d2/de8/group__core__array.html#ga0002cf8b418479f4cb49a75442baee2f)