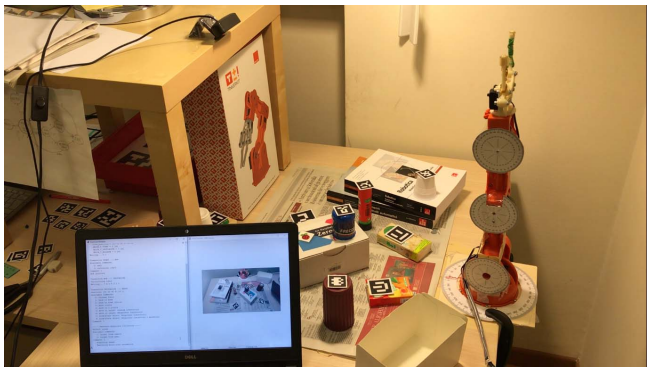
# RoboEye: enable hand-eye coordination with TinkerKit Braccio Robot and ArUco markers

Claudio Verardo, Mattia Balutto, Diego Perissutti[§]
Dipartimento Politecnico di Ingegneria e Architettura (DPIA)
Università degli Studi di Udine
{lastname.firstname}@spes.uniud.it

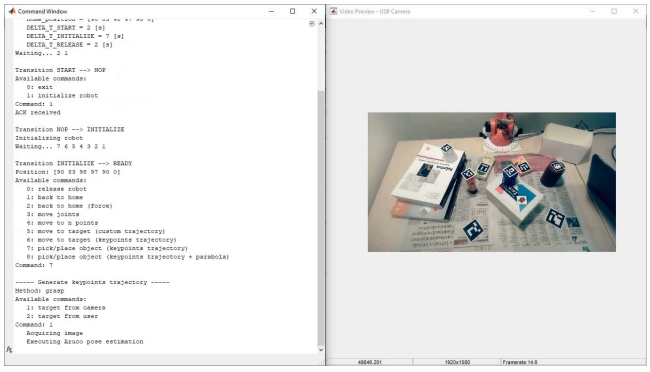
***Abstract*—This project implements a basic hand-eye coordination system between a UVC camera and the TinkerKit Braccio Robot. It is composed of two main modules. The former is a vision pipeline that detects and estimates the poses of the ArUco markers in the scene. The latter is a trajectory planner that solves the inverse kinematics problem in order to reach a desired position and orientation of the end effector.**

**A control module allows the user to interactively define some tasks for the robot, such as moving to a target or pick and place an object identified by a marker. It automatically invokes the vision and the trajectory planning modules when necessary. Moreover, it sends the control signals and the data required by the low-level controller of the robot (Arduino). A calibration module allows the user to calibrate the intrinsics and extrinsics parameters of the camera employed in the setup.**

***Index Terms*—Hand-eye coordination, robotics, vision.**



*(a)* `https://www.youtube.com/watch?v=Vw66SZN9R2s`



*(b)* `https://www.youtube.com/watch?v=31yvqWvIydo`

*Figure 1. Demo videos of the implemented setup from a) an external view and b) the same view of the robot.*

## I. Introduction

This project is the synthesis of what learned during the courses of Robotics and Computer Vision taught by Professors S. Miani, A. Gasparetto, and A. Fusiello at the University of Udine in the academic year 2020-2021.

Our intent was to build from the ground up a basic hand-eye coordination system between a robotic arm and a camera. The main design criterion was to retain the best tradeoff between high accuracy and low hardware costs. Each decision was undertaken in order to achieve this objective.

In this report, we provide an overview of the work carried out. The implementation as well as the results obtained are critically analyzed. All the code, along with a comprehensive documentation, is available in the repository

`https://github.com/claudioverardo/roboeye`

Figure 1 links two demo videos of the developed setup. They show the hand-eye coordination system in action during a pick-and-place experiment with six objects.

As main references, we relied on [1, 2] for computer vision and [3, 4] for robotics backgrounds respectively. Further references will be provided hereafter in the report.

### A. Workflow

Despite the pandemic situation, the work proceeded seamlessly thanks to the use of the git versioning system and a repository hosted on GitHub. The Gitflow paradigm was used throughout the software development with the aim of ensuring modularity and reusability of the code. Matlab was chosen as main programming language in order to allow a fast and flexible prototyping of different approaches to the problem.

### B. Report structure

The report is structured as follow. First, in Section II, we describe the hardware employed in our setup.

Second, in Sections III, IV, V, VI, we present the software developed, which consist of four main modules: robot vision, robot trajectory planning, robot control, robot calibration. Both the underlying algorithms and their implementation are thoroughly commented and motivated.

Finally, in Section VII, we provide both quantitative and qualitative evaluations of the performance of the setup through experiments, and in Section VIII, we draw some conclusions.

[§]All the authors have contributed equally to the project.

## II. Hardware Setup

- **Roffie UC20 webcam 1080p**: a commercial pc webcam, compliant with the USB video class (UVC) standard. This latter feature, allow us to use MATLAB to seamlessly acquire images through the package "MATLAB Support Package for USB Webcams".
- **Arduino Uno**: an open-source micro-controller board based on the Microchip ATmega328P microcontroller. It is equipped with 14 digital I/O pins (six capable of PWM output) and 6 analog I/O pins.
- **Tinkerkit Braccio Robot**: a 5-DOF robotic arm endowed with a gripper as end-effector and controlled via Arduino. It is composed by several bundles:
  - Mechanical bundle: plastics, screws, flat washers, hexagon nuts and springs.
  - Servo motors: 2 x SR 311, 4 x SR 431. Five for the robot joints and one for the end-effector gripper.
  - Power supply: 5V 4A DC.
  - Driving shield: board to supply the servo motors, compatible with Arduino.

  Each servo motor is composed by four main parts: an electrical DC motor, a mechanical gearbox, a feedback potentiometer connected directly to the output shaft of the gearbox, and an integrated chip responsible for driving and controlling the motor in closed-loop.

## III. Robot Vision

The vision module is composed by a pipeline that receives an image as input, spots candidates regions of interest (ROIs), matches them with a dictionary of ArUco markers and estimates their poses in space. It consists of 4 steps:

- **ROI extraction**: the first step extracts the contours from the input image deploying either the adaptive thresholding + Moore-Neighbor tracing or the Canny edge detector + depth-first search (DFS).
- **ROI refinement**: the second step selects only the contours with quadrilateral shapes and refines them in order to identify their corners. To this end, it resorts to either the Ramer–Douglas–Peucker algorithm or a geometric corner extractor. The output are the ROIs candidated for the matching with the ArUco markers.
- **ROI matching**: the third step removes the perspective distortion of the ROIs estimating a proper homography transformation. Then, it looks for matches within the ArUco dictionary exploiting the Hamming distance 2D.
- **ROI pose estimation**: the fourth step estimates the poses in space of the matched ArUco markers through the Perspective-n-Points (PnP) algorithm.

This pipeline takes inspiration from the detection process proposed in the original ArUco library [5], used in well established computer vision libraries, such as OpenCV [6].

Hereafter, we assume the camera to be calibrated, i.e., with known intrisics matrix $\mathbf{K}$, radial distortion coefficient $k_1$, and extrinsics $\mathbf{R}_{\text{cam}}, \mathbf{t}_{\text{cam}}$ with respect the world frame. These parameters can be retrieved for instance with the calibration module discussed in Section VI.

Figure 2a shows the flowchart of the overall pipeline while Figure 5 the step-by-step output obtained with the two images and the ArUco dictionary of Figures 3, 4 respectively.

In the Subsections III-A, III-B, III-C, III-D we present the details regarding the ROI extraction, ROI refinement, ROI matching, and ROI pose estimation steps respectively.

### A. ROI extraction

The aim of the ROI extraction step is to find a set of region of interests in the input image, along with their contours. Figure 2b provides a flowchart representation of this step while Figure 5a shows two output examples.

First, the input image undergoes a grayscale conversion. Then, two different sub-pipelines are available to extract the contours, based on different strategies. They are denoted with "adaptth-moore" and "canny-dfs" respectively.

**Method "adaptth-moore"**

First, the **Adaptive thresholding** algorithm is applied to the image. In details, for each pixel $P$ of the image:

1) Compute a local first-order statistics $\mu$ of the grayscale intensities of the $P$'s neighborhood.
2) Compute the local threshold to $\delta = (1.6 - \sigma)\mu$, where $\sigma \in (0, 1)$ is a sensitivity factor.
3) Set $P$ to 1 if its intensity is above $\delta$, and to 0 otherwise.

The first-order statistics can be for instance the mean, the Gaussian weighted mean, or the median, and it is specified via the adaptth_statistic parameter. Moreover, the parameters adaptth_sensitivity and adaptth_neighborhood set the value of $\delta$ and the size of the pixel neighboorhood. Figure 6a shows an output example of adaptive thresholding.

Second, the **Moore-Neighbor tracing** algorithm is used. This algorithm is designed with the purpose of finding the boundaries of the black (or white) regions in a binarized image, such as the output of the adaptive thresholding. It proceeds as follow (cf. Figure 7c):

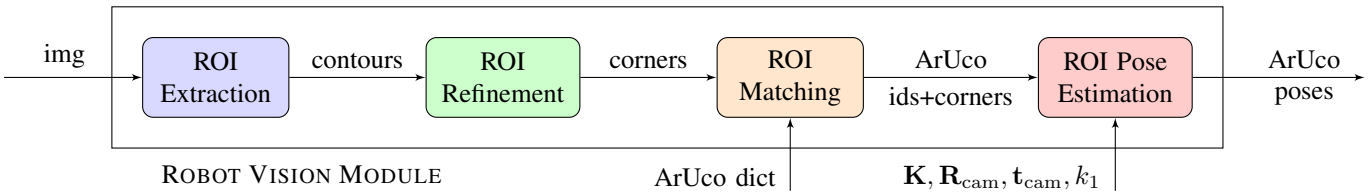
1) Traverse the binarized image. If you find a black pixel, set it to $S$ (start of the boundary).
2) Set the current pixel as $P$ and add it to the current boundary points list.
3) Explore the pixels in the neighborhood of $P$ in clockwise order. If you find a black pixel, set it as new $P$.
4) Repeat the steps from 2 to 3 until $P$ is equal to $S$ with the same orientation (**Jacob's stopping criterion**).
5) Repeat from 1 until the end of the image.

The complexity of this algorithm is $\mathcal{O}(N)$ where $N$ is the sum of all the pixel lengths of the boundaries.

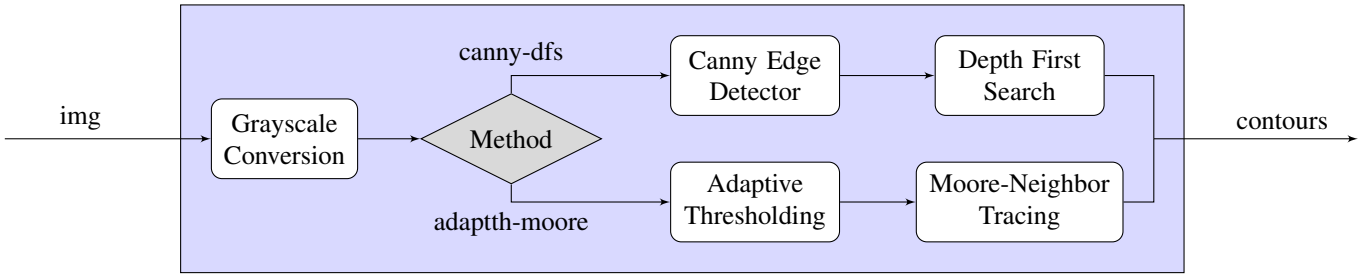
**Method "canny-dfs"**

First, the **Canny Edge Detector** [7] is applied to the image. The main purpose of this algorithm is to highlight the edges within the input image. It proceeds as follows:

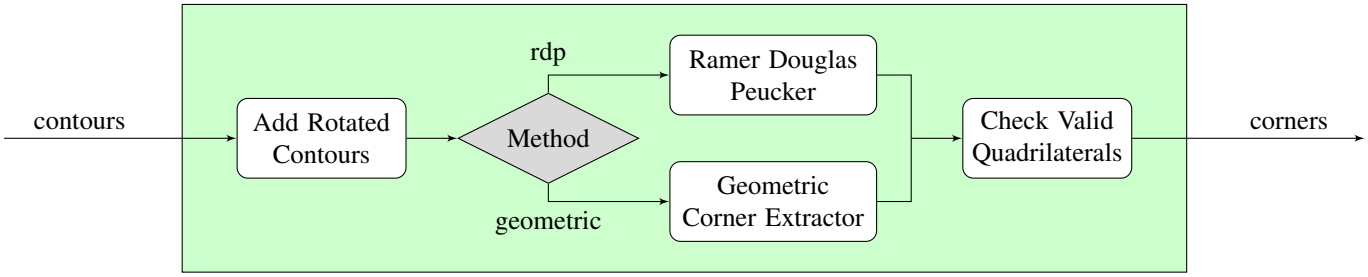
1) Apply a Gaussian filter to smooth the image.



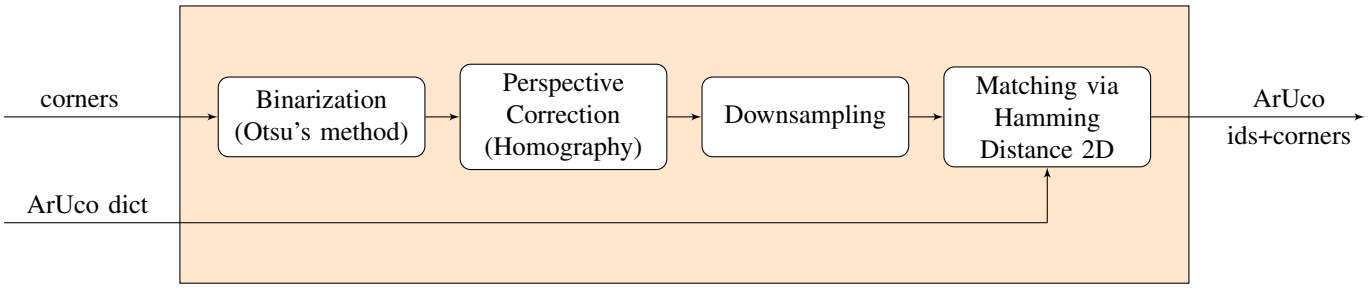

*(a) Flowchart of the vision pipeline.*



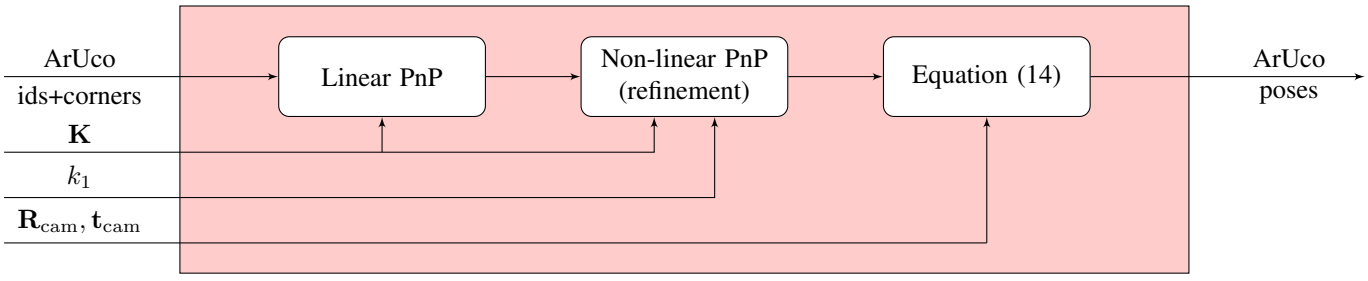

*(b) Flowchart of the ROI extraction step.*




*(c) Flowchart of the ROI refinement step.*




*(d) Flowchart of the ROI matching step.*




*(e) Flowchart of the ROI pose estimation step.*

*Figure 2. Flowchart of the vision pipeline implemented in the Robot Vision module. Figure 2a shows the main blocks of the pipeline while the Figures 2b, 2c, 2d, 2e provides more details regarding the ROI extraction, ROI refinement, ROI matching, ROI pose estimation blocks respectively. Refer to Section III for more details.*

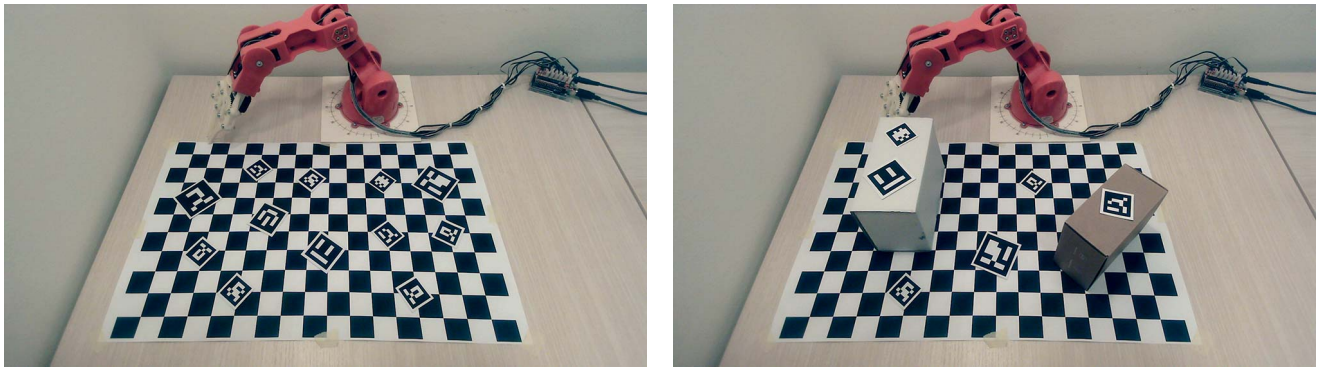


*Figure 3. Images used to the test the vision pipeline. The results are shown in Figure 5.*

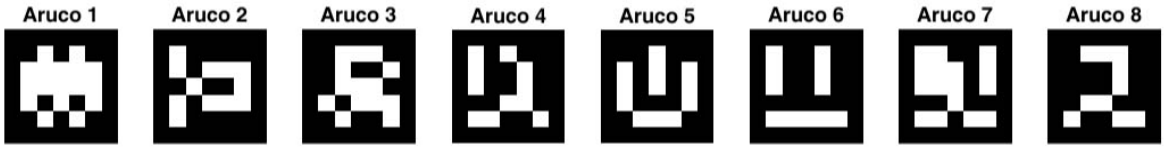



*Figure 4. Aruco dictionary used to the test the vision pipeline. The results are shown in Figure 5.*

2) Find the four intensity gradients of each pixel (along the horizontal, vertical, and diagonal directions).
3) Apply a double threshold to each pixel to determine potential edges. Namely, if the maximum of the four gradients is below the low threshold, discard the pixel. If it is in the middle between the high and low threshold, mark the pixel as weak edge. If it is higher than the high threshold, mark the pixel as strong edge.
4) Track edge by hysteresis. Namely, keep only the pixels that are strong, or weak but nearby other strong pixels.

The low and high threshold are set through the parameters canny_th_low and canny_th_high respectively. Figure 6b shows an example of output.

Second, the **Depth First Search (DFS)** [8] is used. Indeed, the output of Canny is a binary image (1 if a pixel is part of an edge, 0 otherwise). We can see this picture as a graph, where each pixel represents a node with eight undirected edges to its eight neighbors pixels (cf. Figure 7a). On this graph we can perform a DFS to find its strong connected components, thereby identifying the contours of the ROIs (cf. Figure 7b). The time complexity of DFS is $\mathcal{O}(V + E)$ where $V$ are the node of the graph (the pixels of the image) and $E$ are the edges of the graph. Furthermore, the space complexity is $\mathcal{O}(V)$.

The Adaptive thresholding, the Moore-Neighbor tracing and the Canny edge detector are implemented via the Matlab built-in functions `adaptth`, `bwboundaries`, and `edge` respectively. The DFS has been implemented from scratch.

### *B. ROI refinement*

After the ROI extraction step, a collection of ROIs with pointwise defined contours have been identified in the image. The aim of the ROI refinement step is to identify their corners and retain only the ROIs with quadrilateral shape. Figure 2c provides a flowchart representation of this step while Figure 5b shows two output examples.

A ROI is processed only if it contains a number of points greater than roi_size_th. Indeed, the ROI extraction step tends to output several spurious contours composed by few points. Discarding them results in a speedup of the pipeline.

Regarding the corner extraction, two different methods are available. They will be denoted with "rdp" and "geometric".

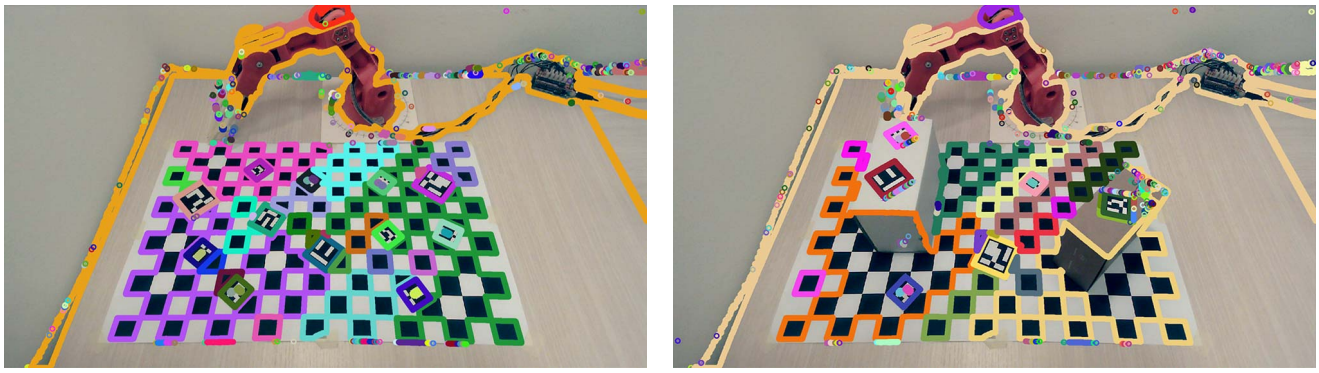
#### Method "rdp"

It builds upon the **Ramer–Douglas–Peucker** algorithm, which recursively decimates an input curve in order to approximate it with a series of line segments. In details:

1) From an ordered input list of points take the start point $S$ and the end point $E$.
2) Check all the euclidean distances between the points in $(S, E)$ and the segment connecting $S, E$.
3) Choose the point $M$ with the maximum distance.
4) Split the line in two disjoint parts, and recursively explore the left sub-part $(S, M)$ and the right sub-part $(M, E)$. Each recursively call continues until the maximum euclidean distance is below a certain threshold called rdp_th ($\epsilon$ in the literature).

In the best case, the complexity of this algorithm is $\mathcal{O}(N \log N)$ while in the worst case, it may achieve $\mathcal{O}(N^2)$. In our code, the algorithm is implemented with the Matlab built-in function `reducepoly`.

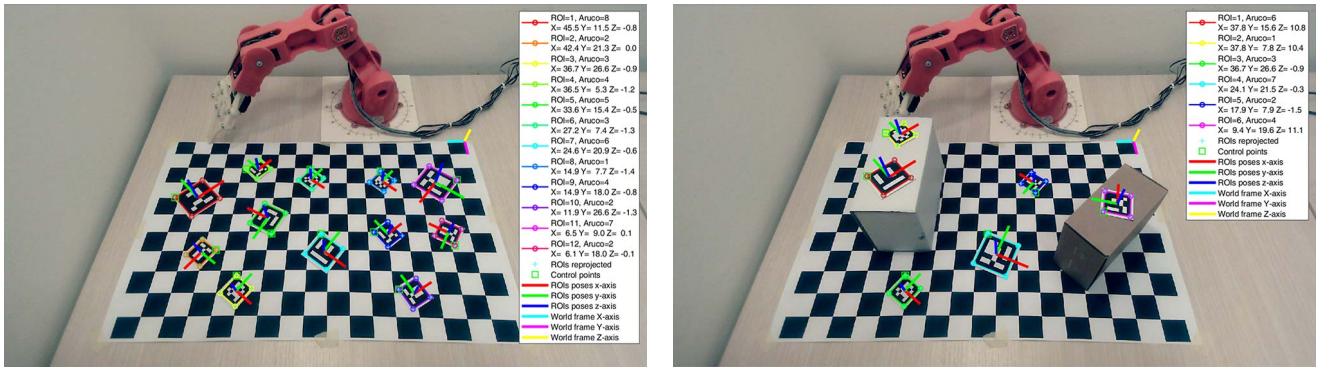
The end points of the output segments are used as corners for the input ROIs. Figure 8a shows the corners of two ROIs retrieved with this method.



*(a) ROI extraction*

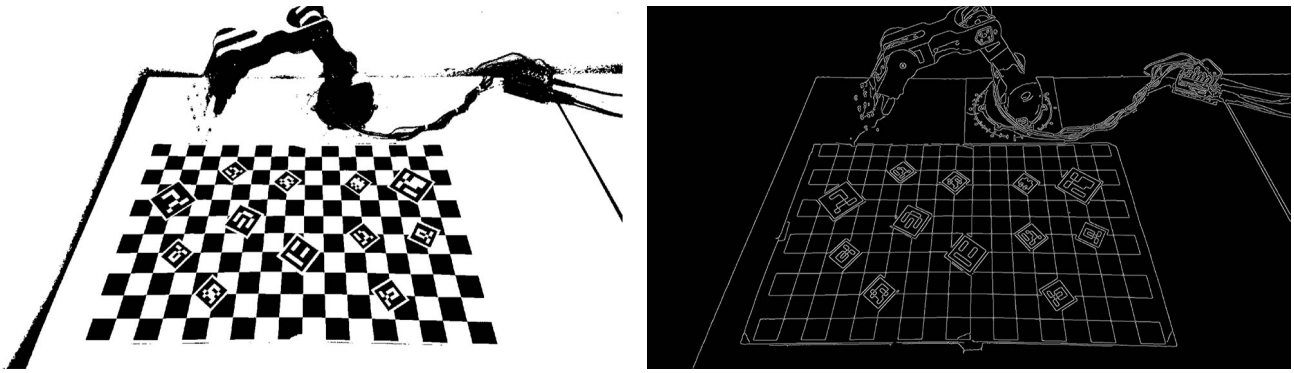
*(b) ROI refinement*

*(c) ROI matching*



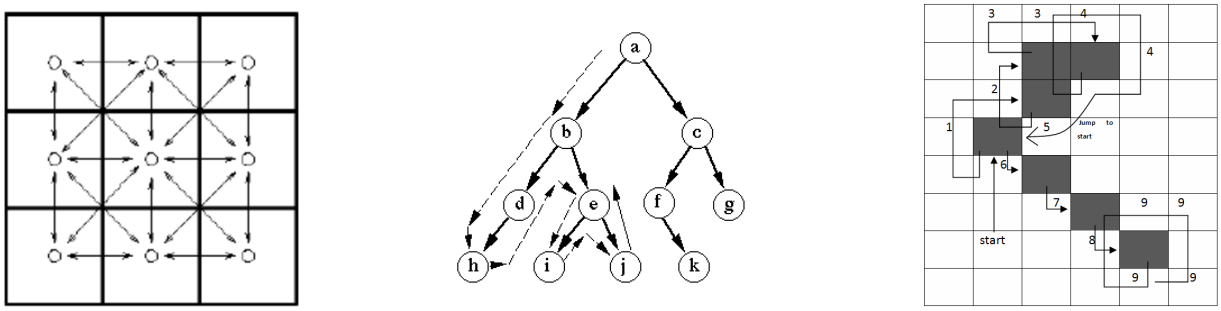
*(d) ROI pose estimation*

*Figure 5. Results obtained at each step of the vision pipeline with the two test images shown in Figure 3 (left and right columns respectively). The dictionary of ArUco markers considered is shown in Figure 4. In ROI extraction, the adaptth-moore method is employed (cf. Subsection III-A). In ROI refinement, the geometric method is employed (cf. Subsection III-B).*



*(a) Adaptive thresholding* *(b) Canny edge detector*

*Figure 6. Comparison between the methods used to perform the ROI extraction.*




*(a) Image becames undirect graph* *(b) DFS* *(c) Moore-Neighbor tracing*

*Figure 7. Graph base approaches used in the ROI extraction sub-pipeline*

**Method "geometric"**

It builds upon a **Geometric corner extractor**, an algorithm that aims to extract from an input curve its four edge corners. We have an array $P$ of $N$ points, where each point is composed by two components ($u = P.u$, $v = P.v$). The four corners $\mathbf{m}_{TL}, \mathbf{m}_{TR}, \mathbf{m}_{BR}, \mathbf{m}_{BL}$ are extracted as follow:

1) Initialize the $\mathbf{m}_{TL}, \mathbf{m}_{TR}, \mathbf{m}_{BR}, \mathbf{m}_{BL}$.
2) Repeat foreach point inside $P$.
3) if $P[i].u < \mathbf{m}_{TL}.u$ or (P[i].u = $\mathbf{m}_{TL}.u$ and $P[i].v < \mathbf{m}_{TL}.v$) then $\mathbf{m}_{TL} = P[i]$.
4) if $P[i].u < \mathbf{m}_{TR}.u$ or (P[i].u = $\mathbf{m}_{TR}.u$ and $P[i].v > \mathbf{m}_{TR}.v$) then $\mathbf{m}_{TR} = P[i]$.
5) if $P[i].u > \mathbf{m}_{BR}.u$ or (P[i].u = $\mathbf{m}_{BR}.u$ and $P[i].v > \mathbf{m}_{BR}.v$) then $\mathbf{m}_{BR} = P[i]$.
6) if $P[i].u > \mathbf{m}_{BL}.u$ or (P[i].u = $\mathbf{m}_{BL}.u$ and $P[i].v < \mathbf{m}_{BL}.v$) then $\mathbf{m}_{BL} = P[i]$.

The complexity of this algorithm is $\mathcal{O}(N)$. Figure 8b shows the corners of two ROIs retrieved with this method.

Concerning the validation process, we discard a shape whenever one of the following conditions is met:

- Number of corners $= 4$.
- Sum of the internal angles $>$ 360 + sum_angles_tol.
- Angle between opposite sides $>$ parallelism_tol.
- Length of a side $<$ side_th_low.
- Length of a side $>$ side_th_high.
- Value of an internal $<$ angle_th_low.
- Value of an internal $>$ angle_th_high.

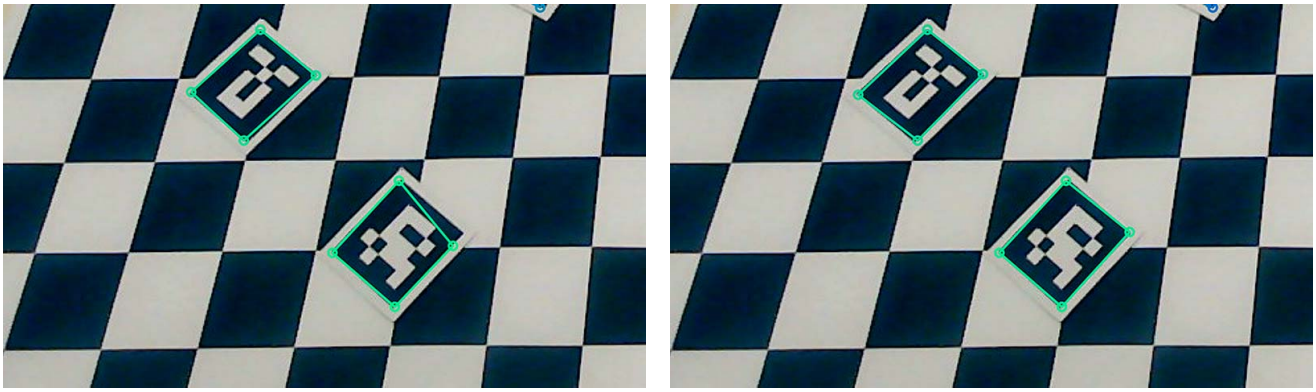
A major issue with the method "geometric" is its lack of robustness in case of shapes with edges parallel to the axes of the image plane. To address it, we propose to consider a replica of the contours rotated by 45 degrees and to feed also them in the ROI refinement procedure. Before the validation process, the rotated ROIs are brought back to their original orientation. After this improvement, we decided to deploy the geometric corner extractor in our experiments because it turned out more reliable to the noise, as it can be seen in the Figure 8.

### C. ROI matching

After the ROI refinement step, a collection of ROIs with quadrilateral shapes along with their corners have been identified in the image. The aim of this step is to find the ROIs that correspond to markers in the input ArUco dictionary. Figure 2d provides a flowchart representation of this step while Figure 5c shows two output examples.

The quadrilateral shape of a ROI is defined by its four corners $\mathcal{M}' = \{\tilde{\mathbf{m}}'_1, \tilde{\mathbf{m}}'_2, \tilde{\mathbf{m}}'_3, \tilde{\mathbf{m}}'_4\}$, where each $\tilde{\mathbf{m}}'_i$ is a point in the image plane[1].

[1]In the following, vectors denoted with and without a tilde will be used to express inhomogeneous and homogeneous coordinates respectively. Namely, by writing $\tilde{\mathbf{v}}$ and $\mathbf{v}$ we mean that $\mathbf{v} = (\tilde{\mathbf{v}}, 1)$.



*(a) Ramer–Douglas–Peucker algorithm* *(b) Geometric corner extractor*

*Figure 8. Comparison between the methods used to perform the ROI refinement. As shown in the images above (cf. the marker on the bottom-right), the Ramer–Douglas–Peucker algorithm tends to retrieve distorted corners for the ROIs. Since this behavior hinders the correct matching of the markers, we deployed the geometric corner extractor in our experiments.*

First, we identify the bounding box of the shape and we binarize its content through the Otsu's method [9].

Second, in order to remove the perspective distortion, we find a proper **homography** that maps $\mathcal{M}'$ into a square shape $\mathcal{M} = \{\tilde{\mathbf{m}}_1, \tilde{\mathbf{m}}_2, \tilde{\mathbf{m}}_3, \tilde{\mathbf{m}}_4\}$, where

$$\begin{aligned} \tilde{\mathbf{m}}_1 &= (0,0) \qquad \tilde{\mathbf{m}}_2 = (0,s) \\ \tilde{\mathbf{m}}_3 &= (s,s) \qquad \tilde{\mathbf{m}}_4 = (s,0) \end{aligned} \tag{1}$$

and $s =$ roi_h_side is an implementation parameter. Thus,

$$\mathbf{m}'_i \simeq \mathbf{H}\mathbf{m}_i \tag{2}$$

where $\mathbf{H}$ is a $3 \times 3$ matrix and the equality holds up to a multiplicative constant. To remove this ambiguity, we consider the cross product between the left and the right terms. Hence,

$$[\mathbf{m}'_i]_\times \mathbf{H}\mathbf{m}_i = \mathbf{0} \tag{3}$$

or equivalently,

$$(\mathbf{m}_i^T \otimes [\mathbf{m}'_i]_\times)\,\mathrm{vec}(\mathbf{H}) = \mathbf{0}\ . \tag{4}$$

Stacking the equations given by each $\mathbf{m}'_i - \mathbf{m}_i$ correspondence, we obtain the system

$$\mathbf{A}\,\mathrm{vec}(\mathbf{H}) = \mathbf{0} \tag{5}$$

where $\mathbf{A}$ is a $12 \times 9$ matrix. Only two of the three equations of (4) are linearly independent. Then, $\mathrm{rank}(\mathbf{A}) = 8$ and the system (5) has a non trivial kernel. A solution is found imposing the component $(\mathbf{H})_{3,3} = 1$. The rectified shape is obtained transforming its bounding box via $\mathbf{H}^{-1}$. In our implementation, we use the built-in Matlab function `fitgeotrans` to estimate the homography.

We then downsample the ROI content to the same size of the markers in the ArUco dictionary (e.g., $7 \times 7$ px).

Finally, we look for matches with the markers in the ArUco dictionary. A match occurs when the **Hamming Distance 2D** between the downsampled ROI and the marker is below a value roi_hamming_th. As candidates for matching we consider all the markers in the ArUco dictionary along with their version rotated by 90, 180, and 270 degrees.

Figure 9 shows two examples of ROI matching. As it can be seen, a parameter roi_hamming_th $> 0$ allows a certain degree of robustness against detection errors of the binary marker.

### *D. ROI pose estimation*

After the ROI matching step, a collection of ROIs along with the correspondent ArUco ids in the dictionary have been identified. The aim of this step is to find the position and orientation of the markers in the world frame. Figure 2e provides a flowchart representation of the ROI pose estimation step while Figure 5d shows two output examples.

First, it solves a linear **Perspective-n-Points (PnP)** problem from 2D-3D correspondences. Indeed, at the four corners of a ROI $\mathcal{M}' = \{\tilde{\mathbf{m}}'_1, \tilde{\mathbf{m}}'_2, \tilde{\mathbf{m}}'_3, \tilde{\mathbf{m}}'_4\}$, correspond the points in space $\mathcal{M} = \{\tilde{\mathbf{M}}_1, \tilde{\mathbf{M}}_2, \tilde{\mathbf{M}}_3, \tilde{\mathbf{M}}_4\}$, expressed in a frame attached to the ArUco. Then, for each 3D-2D correspondence $\tilde{\mathbf{m}}'_i - \tilde{\mathbf{M}}_i$ we obtain the relation

$$\mathbf{m}'_i \simeq \mathbf{K}[\mathbf{R}\ \mathbf{t}]\mathbf{M}_i\ , \tag{6}$$

where $\mathbf{K}$ is the intrinsics matrix of the camera (known from calibration) and $\mathbf{R}, \mathbf{t}$ is the rototranslation that maps points from the ArUco frame into the camera frame (unknown). Since the points lie on a plane, we can suppose without loss of generality the $z$-axis of the ArUco frame to be perpendicular to the plane. Namely, $\tilde{\mathbf{M}}_i = (\tilde{\mathbf{m}}_i, 0)$ where

$$\begin{aligned} \tilde{\mathbf{m}}_1 &= (-l/2, +l/2) \qquad \tilde{\mathbf{m}}_2 = (+l/2, +l/2) \\ \tilde{\mathbf{m}}_3 &= (+l/2, -l/2) \qquad \tilde{\mathbf{m}}_4 = (-l/2, -l/2) \end{aligned} \tag{7}$$

and $l$ is the length of the side of the ArUco[2]. Then, writing $\mathbf{R} = [\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{r}_3]$ the Equation (6) reduces to

$$\mathbf{m}'_i \simeq \mathbf{K}[\mathbf{r}_1\ \mathbf{r}_2\ \mathbf{t}]\mathbf{m}_i\ . \tag{8}$$

[2]For simplicity, we assume that there exist a $1:1$ map between the ArUco ids and the length of their sides in the input ArUco dictionary.

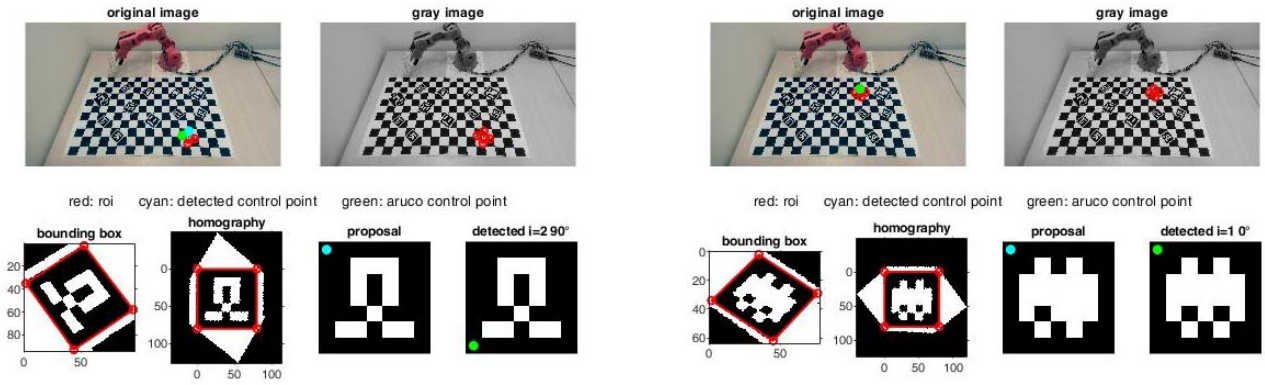



*Figure 9. Details of the ROI matching step for two refined ROIs. The markers proposed for matching are denoted with "proposal", while the correspondent markers identified in the ArUco dictionary are denoted with "detected".*

This relation has the same form of (5) with

$$\mathbf{H} = \lambda \mathbf{K}[\mathbf{r}_1 \ \mathbf{r}_2 \ \mathbf{t}] \ , \tag{9}$$

where $\lambda$ is an unknown multiplicative factor. We can estimate the homography $\mathbf{H}$ from the four correspondences $\mathbf{m}'_i - \mathbf{m}_i$ with the same method described in Subsection III-C. The factor $\lambda$ is retrieved imposing the orthonormality of the matrix $\mathbf{R}$. Namely, writing $\mathbf{H} = [\mathbf{h}_1 \ \mathbf{h}_2 \ \mathbf{h}_3]$ we impose $\lambda = \|\mathbf{h}_1\|_2^{-1}$. Then, from inspection of Equation (9),

$$\begin{aligned} \mathbf{r}_1 &= \lambda \, \mathbf{K}^{-1}\mathbf{h}_1 \\ \mathbf{r}_2 &= \lambda \, \mathbf{K}^{-1}\mathbf{h}_2 \\ \mathbf{t} &= \lambda \, \mathbf{K}^{-1}\mathbf{h}_3 \end{aligned} \tag{10}$$

and moreover $\mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2$. In practice, the matrix $\mathbf{R} = [\mathbf{r}_1 \, \mathbf{r}_2 \, \mathbf{r}_3]$ is not a true rotation matrix, due to noise and not perfect correspondences. We can project it onto SO(3) considering its SVD $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ and imposing

$$\mathbf{R} = \mathbf{U} \ \mathrm{diag}(1, 1, \det(\mathbf{V}^T\mathbf{U}))\, \mathbf{V}^T \ . \tag{11}$$

A further refinement of the estimated rototranslation $\mathbf{R}, \mathbf{t}$ is obtained solving a non-linear PnP problem. Namely, using the output of the linear PnP as starting point, it iteratively minimizes the **reprojection error** of the 2D-3D correspondences

$$\min_{\mathbf{R},\mathbf{t}} \sum_i \| f_d(f_p(\mathbf{K}[\mathbf{R}\,\mathbf{t}]\mathbf{M}_i)) - \tilde{\mathbf{m}}'_i \|_2^2 \ . \tag{12}$$

The function $f_p$ applies the perspective division that converts homogenous coordinates into inhomogenous coordinates $f_p(x, y, z) = (x/z, y/z)$. The function $f_d$ applies the radial distortion, that maps an undistorted point $(u, v)$ in the image plane into a distorted point $(\hat{u}, \hat{v})$ according to[3]

$$\begin{cases} \hat{u} = (1 + k_1 r_d^2)(u - u_0) + u_0 \\ \hat{v} = (1 + k1 r_d^2)(v - v_0) + v_0 \ , \end{cases} \tag{13}$$

[3]More distortion coefficients $k_1, \ldots, k_n$ corresponding to the terms $r_d^2, \ldots, r_d^{2n}$ can be considered in $f_d$. However, in our experiments, we did not appreciate any sensible improvements using more than one coefficient.

where $r_d^2 = (\frac{u-u_0}{\alpha_u})^2 + (\frac{v-v_0}{\alpha_v})^2$. The other parameters are known from the calibration of the camera (cf. Section VI). Indeed, $k_1$ is the radial distortion coefficient, $(u_0, v_0)$ is the principal point, and $\alpha_u, \alpha_v$ the focal length along the $u, v$ axes. Equation (11) is a nonlinear least square problem. It is solved iteratively with the Levenberg-Marquardt algorithm. It requires an expression of the Jacobian $\mathbf{J}$ of the reprojection error with respect to $\mathbf{t}$ and the parameterization of $\mathbf{R}$. For sake of readability, we do not report the expression for $\mathbf{J}$.

Finally, the rototranslation $\mathbf{R}_W, \mathbf{t}_W$ that maps points from the ArUco frame into the world frame can be obtained with the camera extrinsics $\mathbf{R}_{\mathrm{cam}}, \mathbf{t}_{\mathrm{cam}}$ as

$$\begin{aligned} \mathbf{R}_W &= \mathbf{R}_{\mathrm{cam}}^T \mathbf{R} \\ \mathbf{t}_W &= \mathbf{R}_{\mathrm{cam}}^T (\mathbf{t} - \mathbf{t}_{\mathrm{cam}}) \ . \end{aligned} \tag{14}$$

We point out that $\mathbf{t}_W$ represents the coordinates of the center of the ArUco with respect the world frame.

## IV. Robot Trajectory Planning

The trajectory planning module provides a tool to generate trajectories for the Tinkerkit Braccio Robot. Namely:

- It computes the direct kinematics of the robot using its Denavit–Hartenberg (DH) parameters.
- It solves the problem of inverse kinematics for a given position and orientation of the end effector in space. Three different approaches are implemented. The first addresses the full problem aiming to solve the direct kinematics equations with respect to the positions of the joints. The second and the third exploit some domain knowledge to reduce the number of joints to be considered from 5 to 3 and 2 respectively. The latter approaches lead to more stable and computationally efficient routines.
- Leveraging the solutions of the inverse kinematics, it allows to retrieve keypoints in the joints space from specifications of the end effector pose. These keypoints are then interpolated into a trajectory.
- When a target position specify an object to be grasped, it allows to adjust the grasping objective with some object-specific offsets in order to guarantee a solid grasp.

- Studying the geometric Jacobian of the robot, it identifies the singularities among the joints positions.
- Monitoring the positions of the joints, it avoids collisions of the robot with the ground.

Figure 10a shows a flowchart representation of this module. In order to use the poses of the ArUco markers as targets, the transformation between the robot and the world reference frame need to be provided. The strategy used to match them is discussed in Subsection VI-C.

In Subsections IV-A, IV-B we present the solutions to the direct and inverse kinematics problems. In Subsections IV-C, IV-D we discuss the heuristic implemented to interpolate trajectories from keypoints and to retrieve the grasping objectives. In Subsection IV-E, IV-F we show how to detect singularities and collisions in the joints space.

### A. Direct kinematics

The direct kinematics algorithm returns the rototranslation matrix of the end effector (EF) with respect to the robot reference frame (RRF) when the angular positions of the joints are given. The origin of RRF is defined at the center of the robot base and the position of the EF is defined as the tip of the gripper (at the center) when it is completely closed (cf. Figure 11). The direct kinematics algorithm allows also to find the position and orientation of a generic joint in the RRF, according to the DH convention.

Table IIa reports the measured DH parameters.

### B. Inverse kinematics

The inverse kinematics returns the positions of the joints that realizes a given pose of the end effector. Three different version of the inverse kinematics are available:

1) The first uses a numerical solver in Matlab to find the joint positions that give the desired EF rototranslation matrix given in input. This version is the most general one and works in principle for all kind of robots, but it is slow and rather unstable.
2) The second is designed specifically for this robot. It takes as a input the position $(x_{EFr}, y_{EFr}, z_{EFr})$ and the second and third Euler angles $\varphi_2, \varphi_3$ of the EF[4]. The positions of the joints 1 and 5 readily follows:
$$\begin{aligned} q_1 &= \operatorname{atan2}(y_{EFr}, x_{EFr}) \\ q_5 &= \varphi_3 \ . \end{aligned} \tag{15}$$
The others are computed numerically solving the system
$$\begin{cases} 0 = \operatorname{sign}(x_{EFr}) \sqrt{x_{EFr}^2 + y_{EFr}^2} \\ \quad + a_2 \sin(q_2) + a_3 \sin(q_2 + q_3) + d_5 \sin(\varphi_2) \\ 0 = a_2 \cos(q_2) + a_3 \cos(q_2 + q_3) \\ \quad + d_5 \cos(\varphi_2) + d_1 - z_{EFr} \\ \varphi_2 = q_2 + q_3 + q_4 \end{cases} \tag{16}$$
For every EF position, except for the singularities, there are up to two joint positions that satisfies inverse kinematics with different elbow position (cf. Figure 12). The inverse kinematics finds only one of the two solutions. Namely, it chooses the solution that tends to have the elbow joint further from the floor (provided that the joints does not exceed the maximum angles).
3) The third method is designed in order to maximize the workspace that the EF is able to reach. The algorithm uses the same principle than the second method, but takes in input only the desired position and the third Euler's angle $\varphi_3$ of the EF. Then, it chooses $\varphi_2$ by itself. Namely, the algorithm scans for all the possible inverse kinematics solutions starting with the one with $q_4 = 0$ and then moving it towards the ground. If no solution that satisfies the joints excursions is found, it scans in the opposite direction. The equations solved are the same as the previous method by imposing $\varphi_2 = q_2 + q_3 + \bar{q}_4$ with $\bar{q}_4$ now fixed (cf. Figure 10b). When a previous joints configuration is provided, the algorithm can also find the closest configuration (in joint space) with the end effector in a given spatial position.

In our experiments, we used the third version.

### C. Keypoints interpolation

Exploiting the inverse kinematics it is possible to define a set of keypoints in the joint space. Each keypoint is in the form $\mathbf{q} = \{q_1, q_2, q_3, q_4, q_5, q_6\}$ where $q_1, \ldots, q_5$ are the solutions of the inverse kinematics, while $q_6$ is the aperture of the gripper. Between each sequential pair of keypoints, a trajectory is generated with the following interpolation strategy:

1) Rotate only the base of 1 degree every timestep[5] delta_t with an overshoot of 5 degree until the desired $q_1$.
2) Rotate all the other joints of 1 degree every timestep delta_t until the desired angular position $q_2, \ldots, q_6$.

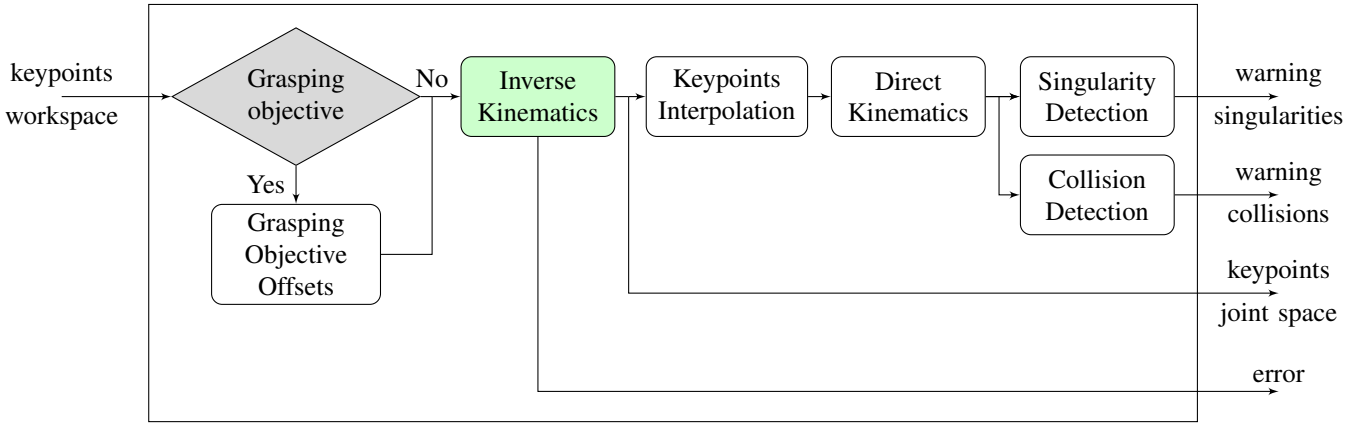
Step 1 and 2 could also be inverted if needed. We experimentally observed that step 1 reduces the severe effects of the first joint's backlashes, thereby improving the precision. Besides, step 2 ensures a motion for the joints with constant velocity of 1 degree/delta_t. This reduces vibrations and further improves the repeatability of the robot. Figures 13 and 14 show an example of interpolated trajectory.

As far as the implementation is concerned, the actual trajectory used to drive the robot is interpolated by the micro-controller within the control module (cf. Subsection V-A). Nevertheless, the same procedure is performed within the trajectory generator tool in order to check singularities and collisions (cf. Subsections IV-E and IV-F).

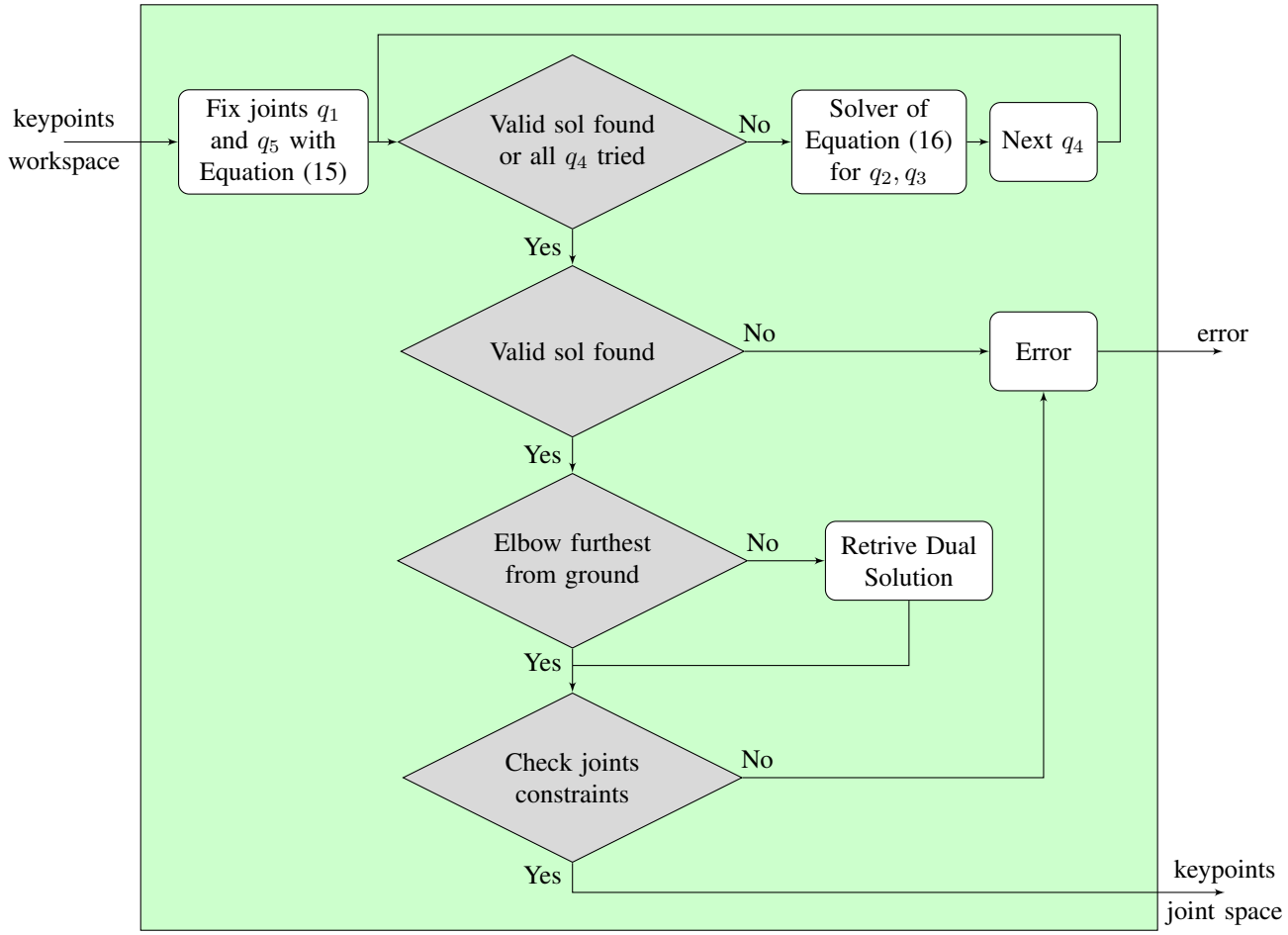
As it will be explained in Subsection V-A it is also possible to define trajectories in a pointwise fashion with our implementation. However, we advocate for trajectories defined by keypoints in our experiments for two main reasons. First, the micro-controller is able to store only a limited number of points (about 200). Second, the robot experiences a lot of vibrations if the trajectory does not guarantee a constant, low velocity profile to each joint.

[4]The angle $\varphi_1$ cannot be chosen because the robot wrist has only 2 DOFs.

[5]We used delta_t =30ms as suggested by the Braccio's manual.

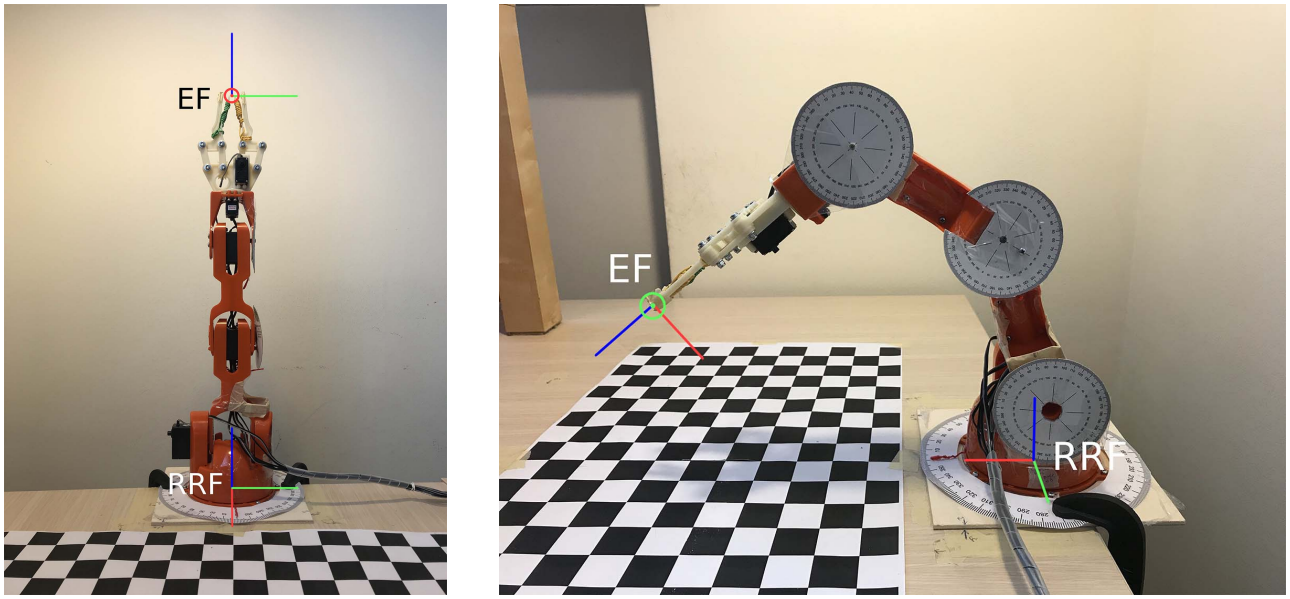



*(a) Flowchart of the trajectory generator tool.*




*(b) Flowchart of the inverse kinematics algorithm (the 3rd version described in Subsection IV-B).*

*Figure 10. Flowchart of the trajectory generator tool implemented in the Robot Trajectory Planning module. Figure 10a shows its main blocks, while Figure10b provides more details regarding the inverse kinematics. Refer to Section IV for more details.*




*(a) Front view* *(b) Side view*

*Figure 11. Robot Reference Frame (RRF) and End Effector (EF) reference frame.*

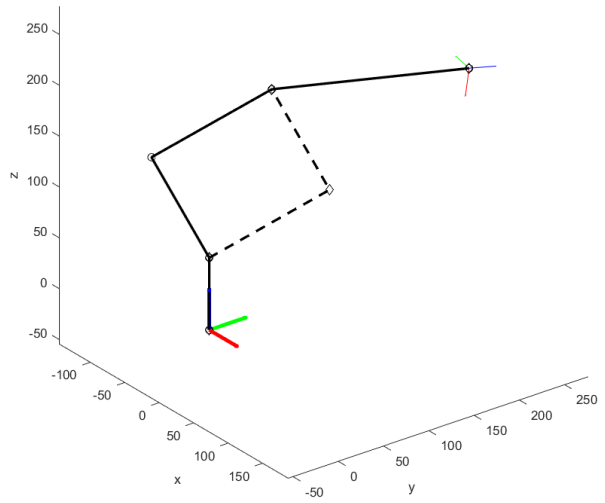


*Figure 12. Two different robot configurations that solve the inverse kinematics problem with the same EF pose.*

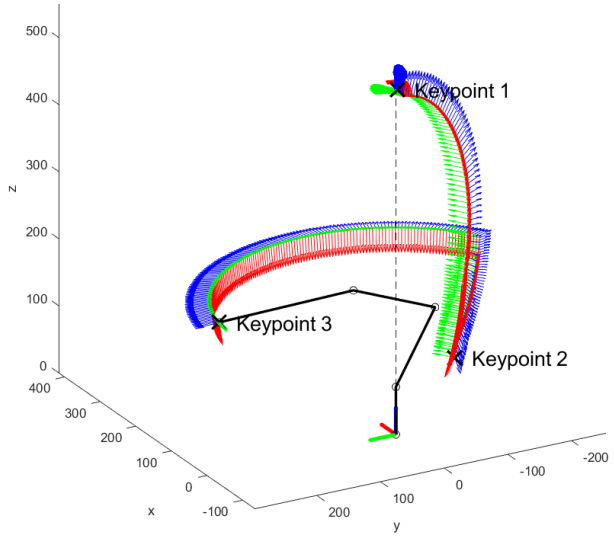



*Figure 13. Example of trajectory interpolated from home position (Keypoint 1) to pick position (Keypoint 2) and place position (Keypoint 3). EF reference frame is displayed for every point of the discrete trajectory and the robot wire-frame is referred to the last point.*

### *D. Grasping objectives*

The main task of the robot is to grasp a target object identified by an ArUco marker attached to it. However, the position of the marker (retrieved by the vision module) is not sufficient to guarantee a solid grip if used as target for the inverse kinematics. For this reason, we implemented a sub-module that allow to adjust the target position of the EF with additional radial and vertical displacements (offset_r and offset_z in the code), which depend on the target object. Moreover, the gripper should not hold the object in its very tip (the EF reference frame origin), but in its inner part. We implemented this by subtracting to the $d_5$ DH parameter a quantity called offset_ef, which also depends on the target object. In this way the inverse kinematics algorithm provides a position that allows the robot to grasp the object in a point located offset_ef mm distant from the gripper tip.

### *E. Singularity detection*

This sub-module detects if a given trajectory encounters singular configurations by means of the geometric Jacobian

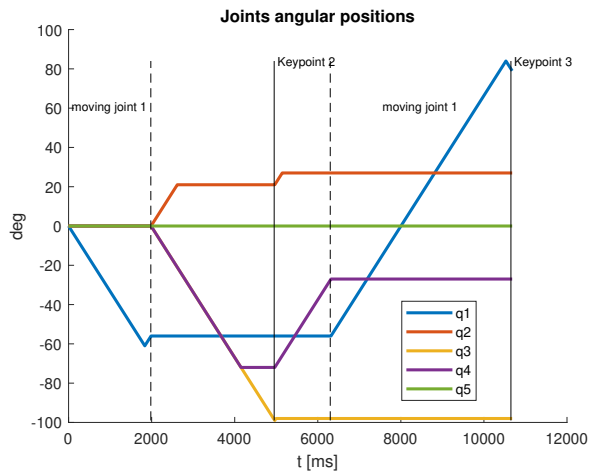



*Figure 14. Joint angular positions over time of the trajectory shown in Figure 13.*

matrix $\mathbf{J}$. For a 5 DOFs robot with only rotoidal joints, the general formula [3] for the $i$-th column of the matrix reads

$$\mathbf{J}_i = \begin{pmatrix} \mathbf{z}_{i-1} \times (\mathbf{p} - \mathbf{p}_{i-1}) \\ \mathbf{z}_{i-1} \end{pmatrix},$$

where $\mathbf{p}$ is the position of EF, $\mathbf{z}_i$ is the versor of the $z$ axis or the RF on the $i$-th joint (in conformity with the DH convention) and $\mathbf{p}_i$ the position of its origin. All these data are computed using the direct kinematics algorithm. The geometric Jacobian $\mathbf{J}$ is computed for every point of the trajectory, and warning messages are arisen if singular configurations are encountered (i.e., $\text{rank}(\mathbf{J}) < 5$).

### F. Collision detection

When a target destination is provided, this sub-module first ensures that the $z$-position is above z_min in order to not collide with the ground. Second, once the trajectory is generated, it warns the user if in certain configurations the joints $q_i$ are below a certain distance joint_safety_radius(i) from the ground. This is accomplished via the direct kinematics.

## V. Robot Control

The control module provides the low-level controller of the robot and its high-level interface. Namely:

- A finite-state-machine (FSM) that runs on Arduino and control the robot behavior through 10 different states and their control signals/data received from the serial port.
- A Matlab interface with the Arduino FSM through the serial connection. It keeps track of the state transitions and allows the user to send control signals and trajectory data from the Matlab command window. The interaction with the vision and trajectory planning modules is handled by the interface itself.

Figure 15a highlights the roles of the control module among the other software. In Subsection V-A, we discuss the Arduino FSM while in Subsection V-B its Matlab interface.

### A. Arduino FSM

Figure 15b shows the complete transition diagram of the Arduino FSM. It consists of 10 different states:

- START: entry state at startup. It initializes the serial connection and send to Matlab some configuration parameters of the robot (e.g., the home position).
- NOP: state of "no operation". The robot is turned off and the micro-controller waits for a control signal.
- INITIALIZE: initialization state for the robot. It switches on the motors and brings the robot to the home position. Then, a transition to READY occurs.
- READY: it sends to Matlab the current position of the robot (cf. Figure 15c) to allow the generation of a trajectory. It then waits for a control signal.
- LOAD TRAJECTORY: it receives the trajectory data sent by Matlab on the serial connection. Then, it triggers a transition to POINTWISE TRAJECTORY or KEYPOINTS TRAJECTORY accordingly. The trajectory data are organized as depicted in Figure 15d.
- POINTWISE TRAJECTORY: the trajectory loaded during LOAD TRAJECTORY is executed in a pointwise fashion[6]. Then, a transition to READY occurs.
- KEYPOINTS TRAJECTORY: the keypoints loaded during LOAD TRAJECTORY are interpolated as described in Subsection IV-C and the resulting trajectory is executed. Then, a transition to READY occurs.
- RELEASE: releasing state for the robot. It switches off the motors. Then, a transition to NOP occurs.
- ERROR STATE: destination of uncontrolled transitions. The robot remains in idle and a LED switches on.
- END: final state, the robot and the serial connection are turned off. They remain in idle until the next reboot.

Each control signals and data received is written back on the serial as acknowledge (ACK). The same applies to transitions.

### B. Matlab interface

During the execution of the Arduino FSM, the Matlab interface guides the user through the functionalities available at each state. The proper control signals are chosen from the command window and the correctness of the state transitions is checked via the ACK messages of the micro-controller.

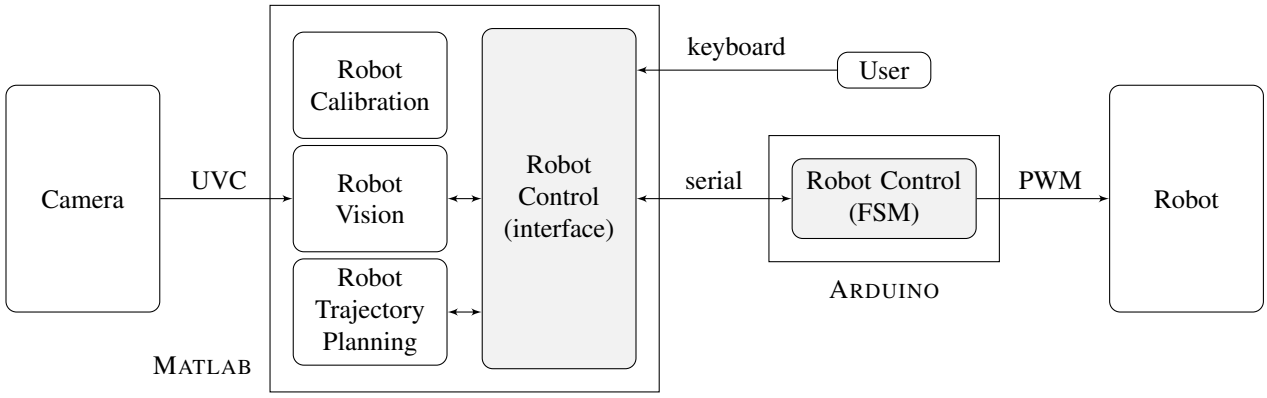
When the robot is in the state READY, the trajectory generator tool provided in trajectory planning module is invoked. It allows to interactively define some basic tasks such as pick-and-place objects or move to targets identified by ArUco markers. The generated trajectory is then sent to Arduino in the format shown in Figure 15d.

## VI. Robot Calibration

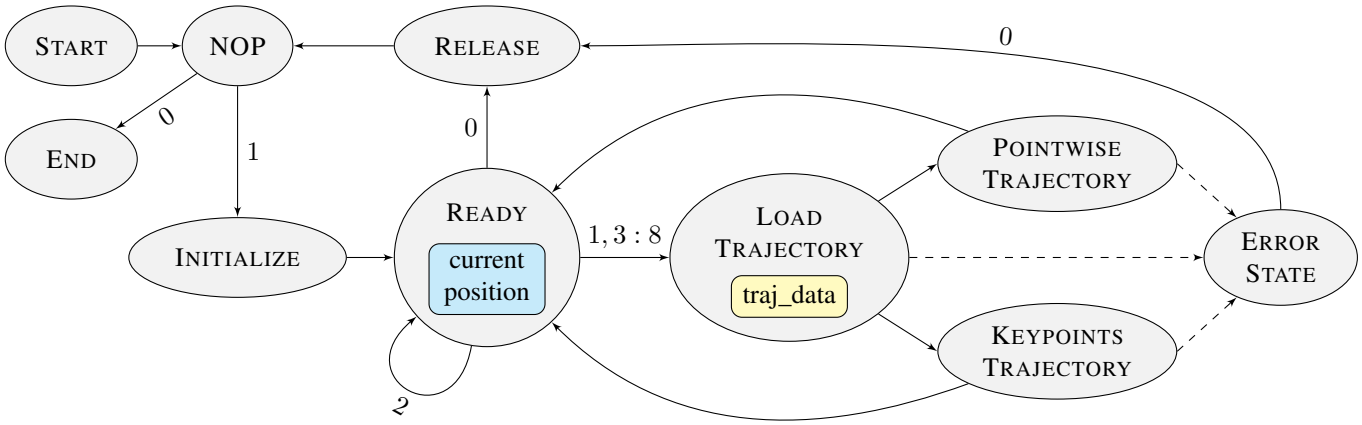
The calibration module provides some utilities to calibrate the camera used by the vision module. Namely:

- Intrinsics and radial distortion calibration via the Sturm-Maybank-Zhang (SMZ) algorithm.
- Extrinsics calibration via the PnP algorithm.

[6]Not used in our experiments, as commented in Subsection IV-C.




*(a) Schematic of the behavior of the Robot Control module. It implements both the finite-state machine (FSM) that is run on Arduino (cf. Figure 15b) and its interface on Matlab. The latter, automatically interacts with the vision and the trajectory planning modules, thus allowing the user to program some basic tasks on the robot, such as reaching a desired position and/or grasp an object.*




*(b) Finite-state machine (FSM) running on the Arduino micro-controller. Nodes represent the states of the FSM, with the exception of* READY *and* LOAD TRAJECTORY*: indeed, they represent a family of possible states parametrized by the variables current_position (cf. Figure 15c) and traj_data (cf. Figure 15d) respectively. Solid edges represent controlled transitions: if a label is present on an edge it represent the control signal that triggers the transition, otherwise it means that the transition is handled internally by the micro-controller. Dashed edges represent uncontrolled transitions that are triggered when an internal error occurs, such as invalid trajectory data in traj_data within* LOAD TRAJECTORY *or an unexpected stop of the trajectory execution within* POINTWISE TRAJECTORY *or* KEYPOINTS TRAJECTORY*.*

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| base_pos | shoulder_pos | elbow_pos | wrist_pitch_pos | wrist_roll_pos | gripper_pos |

*(c) Angular positions of the robot joints stored in the Arduino micro-controller. The values are updated at the end of the execution of each trajectory, namely when a transition from either* POINTWISE TRAJECTORY *or* KEYPOINTS TRAJECTORY *to* READY *occurs.*

| 0 | 1 | 2 | 3 | $\cdots$ | 8 | $\cdots$ | $6N-3$ | $\cdots$ | $6N+5$ |
|---|---|---|---|---|---|---|---|---|---|
| traj_type | $N$ | delta_t | traj$[1,1]$ | $\cdots$ | traj$[1,6]$ | $\cdots$ | traj$[N,1]$ | $\cdots$ | traj$[N,6]$ |

*(d) Serial data sent to Arduino when a transition from* READY *to* LOAD TRAJECTORY *occurs. Each cell correspond to 1 byte, namely: traj_type equal to 1 (2) denotes a pointwise (keypoints) trajectory, $N$ is the number of points (keypoints) of the trajectory, delta_t is the timestep associated to each point of the trajectory, and traj$[i, j]$ is the position of the $j$-th joint in correspondence of the $i$-th point (keypoint) of the the trajectory. A value of traj_type equal to 1 (2) triggers a transition from the state* LOAD TRAJECTORY *towards the state* POINTWISE TRAJECTORY *(*KEYPOINTSTRAJECTORY*). Otherwise, a transition to* ERROR STATE *occurs.*

*Figure 15. Details of the Robot Control module. Refer to Section V for further information.*

Both the two procedures require a checkerboard pattern with known dimensions of its squares. It is thus possible to define a frame attached to the checkerboard and assign a set of coordinates to each corner within the checkerboard.

In Subsections VI-A, VI-B we present the calibration procedures for the intrinsics and the extrinsics respectively. Regarding the calibration between the world and the robot frames, an automatic routine is not available. However, in Subsection VI-C, we explain a simple strategy to perform it.

### A. Camera intrinsics calibration

This procedure allows to retrieve the intrinsics matrix $\mathbf{K}$ and the radial distortion coefficient $k_1$ of the camera.

First, it acquires at least three images of the checkerboard pattern that lie on different planes. Then, it runs the **Sturm-Maybank-Zhang (SMZ)** calibration algorithm [10, 11]. The underlying idea is to resect the homographies induced by the planes with a QR decomposition and then refine the estimates of the parameters with bundle adjustement. Our code is a binding of the SMZ implementation offered in the Calibration Toolbox by Andrea Fusiello [12].

Table IIb reports the parameters obtained for the camera employed in the experiments. Figure 16a shows two example of output obtained during the calibration.

### B. Camera extrinsics calibration

This procedure allows to retrieve the extrinsics parameters $\mathbf{R}_{\text{cam}}, \mathbf{t}_{\text{cam}}$ of the camera. They represent the rototranslation that maps the coordinates of a point from the world frame into the camera frame, where the world frame is the one attached to the checkerboard. The intrinsics $\mathbf{K}$ and the radial distortion coefficient $k_1$ are assumed to be known.

First, an image of a fixed checkerboard is acquired and it is asked to the user to identify its four corners in the image plane. Second, the perspective distortion is removed computing a proper homography transformation, with the same procedure described in Subsection III-C. Then, the corners within the checkerboard are identified with a pattern-matching strategy. Finally, the extrinsics $\mathbf{R}_{\text{cam}}, \mathbf{t}_{\text{cam}}$ are estimated from the 2D-3D correspondences of coordinates of the corners in the image plane and in the world frame, solving a PnP problem with the same procedure described in Subsection III-D.

Figure 16b shows an output example with the world frame fixed at the top right corner of the checkerboard.

### C. Robot-camera calibration

First, fix the base of the robot in a desired position and place the checkerboard in order to have one point $O$ with known position with respect the center of the robot base. Second, rotate the checkeboard keeping $O$ fixed, and align it with the axes of the RRF.

## VII. Experiments

In order to evaluate the performances of the implemented system, we carried out both quantitative and qualitative experiments. In Subsection VII-A, we measures the accuracy of the vision pipeline in estimating the poses of the markers. In Subsection VII-B, we show the capabilities of the overall system in accomplish a vision-driven grasping experiment.

Table III reports the parameters used throughout the tests.

### A. Vision module evaluation

To evaluate the accuracy of the vision module, a set of ArUco with known poses in space were needed. Thus, we generated 20 images of a checkerboard with superimposed markers in different positions and orientations (cf. Figure 17a) through a computer graphics software. Then, by projecting the images on a fixed screen (cf. Figure 17b) it was possible to quantitatively compare the estimates of the poses obtained from the vision module with their real values.

In details, first the camera was calibrated with respect the checkerboard with the procedure described in Subsection VI-B. Then, the estimates of the poses retrieved by the vision pipelines[7] "adaptth-moore" and "canny-dfs" were collected. Table I shows the results.

The estimation turned out to be very precise for the positions $t_x, t_z$ of the markers along the $x, y$-axes, while the error on the $z$-coordinate $t_z$ is roughly an order of magnitude higher. Also the estimation of the orientation[8] $\theta$ shows sound performances. The two pipelines takes nearly the same amount of time. The computational bottleneck is the ROI extraction step, which deals with a processing of the entire image.

### B. Grasping of objects

In this experiment, the task was to pick one or more objects in the workspace and put them in a box. For each object, the task was achieved using the keypoints interpolation described in Subsection IV-C for the following keypoints

1) Current position.
2) Object position (gripper open).
3) Object position (gripper closed).
4) Box position (gripper closed).
5) Box position (gripper open).

Each object is identified by a different ArUco marker stuck on the top of it. For every marker (i.e., for every object) we defined a proper value for the offsets introduced in Subsection IV-D. In this way, the vision system was able to autonomously recognise the objects and to perform the grasps.

The videos linked in Figure 1 show the robot performing 6 successful grasps. The following videos show the same task with a slightly different arrangement of the objects:

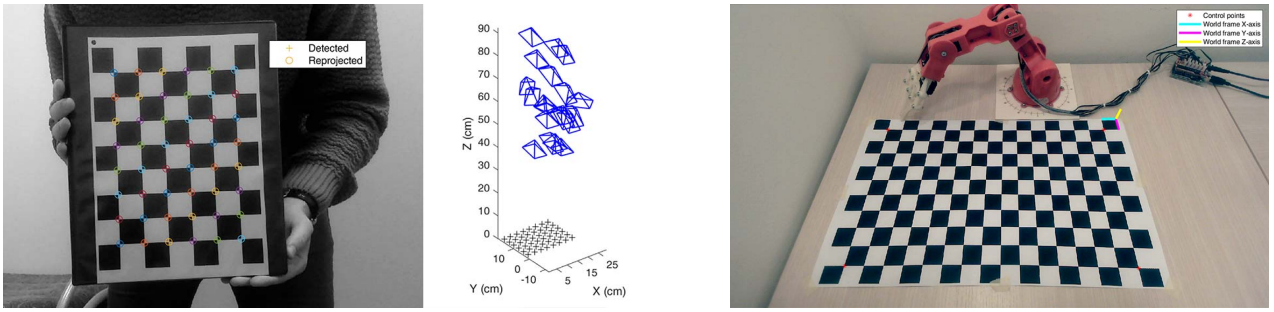
- `https://www.youtube.com/watch?v=Kzpq9sqbxM0`
- `https://www.youtube.com/watch?v=rr2VxXzEknk`

## VIII. Conclusion

Despite the low cost hardware employed, we claim that we have reached a decent performance with our hand-eye coordination system. The experiments showed that the precision provided by the vision system, the trajectory planning and the

[7]The names stand for the method used in the ROI extraction step.

[8]The orientation error $\theta$ is extracted first aligning the estimated $z$-axis with the ground truth one. Then, calculating the angle between the two $x$ axes.




*(a) Intrinsics calibration* *(b) Extrinsics calibration*

*Figure 16. Calibration procedures of the camera implemented in the calibration module.*

*(a)* *(b)*

*Figure 17. Example of image generated for the vision pipeline evaluation.*

| | adaptth-moore | canny-dfs |
|---|---|---|
| reprojection error lin [px] | $1.18 \pm 0.88$ | $1.38 \pm 1.53$ |
| reprojection error nonlin [px] | $0.23 \pm 0.14$ | $0.26 \pm 0.18$ |
| position error $t_x$ [cm] | $0.10 \pm 0.16$ | $0.09 \pm 0.15$ |
| position error $t_y$ [cm] | $0.08 \pm 0.16$ | $0.09 \pm 0.17$ |
| position error $t_z$ [cm] | $-1.13 \pm 0.39$ | $-1.02 \pm 0.38$ |
| position error $\|t\|_2$ [cm] | $1.16 \pm 0.39$ | $1.04 \pm 0.39$ |
| orientation error $\theta$ [deg] | $-0.53 \pm 0.79$ | $-1.00 \pm 1.34$ |

| | adaptth-moore | canny-dfs |
|---|---|---|
| time ROI extraction [s] | $1.07 \pm 0.06$ | $1.34 \pm 0.19$ |
| time ROI refinement [s] | $0.02 \pm 0.003$ | $0.13 \pm 0.22$ |
| time ROI matching [s] | $0.53 \pm 0.07$ | $0.21 \pm 0.18$ |
| time ROI pose estimation [s] | $0.03 \pm 0.01$ | $0.03 \pm 0.01$ |
| time total [s] | $1.64 \pm 0.11$ | $1.71 \pm 0.55$ |
| # ROIs extracted | $292 \pm 20$ | $388 \pm 16$ |
| # ROIs refined | $37 \pm 3$ | $5 \pm 1$ |

*Table I. Results of the evaluation tests for the vision module. Data are expressed as mean and standard deviations.*

robot actual movements were enough to accomplish simple pick and place operations of different objects reliably, and in an almost completely autonomous fashion.

Possible future works to improve the system may concern the grasping of objects that require a non-horizontal gripper orientation, for example by adjusting the gripper orientation with the ArUco orientation. Another improvement, could be the porting of all the software in C++ in order to make it faster and, possibly, to achieve real-time computations.

## REFERENCES


[1] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, 2004.

[2] R. Szeliski, *Computer Vision: Algorithms and Applications*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 2010.

[3] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2010.

[4] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 1st ed. Springer Publishing Company, Incorporated, 2013.

[5] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[6] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[7] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[9] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[10] P. F. Sturm and S. J. Maybank, "On plane-based camera calibration: A general algorithm, singularities, applications," in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, vol. 1, 1999, pp. 432–437 Vol. 1.

[11] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[12] A. Fusiello, "Calibration toolkit," [Online]. http://www.diegm.uniud.it/fusiello/demo/toolkit/calibration.html.

## APPENDIX

| Joint | $a$ [mm] | $d$ [mm] | $\alpha$ [deg] | $\theta$ [deg] |
|---|---|---|---|---|
| 1 | 0 | 71 | 90 | $q_1$ |
| 2 | 125 | 0 | 0 | $q_2$ |
| 3 | 125 | 0 | 0 | $q_3$ |
| 4 | 0 | 0 | -90 | $q_4$ |
| 5 | 0 | 195 | 0 | $q_5$ |

*(a)*

| Parameter | Value | Unit | Description |
|---|---|---|---|
| $\alpha_u$ | -1000.4 | px | focal length along u-axis |
| $\alpha_v$ | -996.5 | px | focal length along v-axis |
| $u_0$ | 971.1 | px | principal point u-coordinate |
| $v_0$ | 538.6 | px | principal point v-coordinate |
| $s$ | 2.54 | px | skew coefficient |
| $k_1$ | 0.0919 | - | radial distortion coefficient |

*(b)*

*Table II. a) DH parameters of the robot, b) calibrated intrinsics and radial distortion coefficient of the camera.*

| **Parameter** | **Default** | **Units** | **Description** |
|---|---|---|---|
| ROI Extraction (cf. Subsection III-A) | | | |
| roi_extraction_method | 'adaptth-moore' | - | choose the ROI extraction algorithm ('adaptth-moore' or 'canny-dfs') |
| adaptth_sensitivity | 0.7 | - | sensitivity of the adaptive thresholding |
| adaptth_statistic | 'gaussian' | - | statistic of the adaptive thresholding ('gaussian', 'mean' or 'median') |
| adaptth_neighborhood | [135 241] | px | neighborhood size of the adaptive thresholding |
| canny_th_low | 0.01 | - | lower threshold of the Canny edge detector |
| canny_th_high | 0.10 | - | higher threshold of the Canny edge detector |
| ROI Refinement (cf. Subsection III-B) | | | |
| roi_refinement_method | 'geometric' | - | choose the ROI refinement algorithm ('rdp' or 'geometric') |
| roi_size_th | 50 | - | min #points required by each ROI to be processed |
| rdp_th | 0.2 | px/ diag(ROI) | threshold of the Ramer-Douglas–Peucker algorithm |
| roi_sum_angles_tol | 10 | deg | tolerance on the sum of the internal angles |
| roi_parallelism_tol | 10 | deg | tolerance on the angle between opposite sides |
| roi_side_th_low | $1/100$ | px/ diag(img) | lower threshold on the length of each side |
| roi_side_th_high | $1/5$ | px/ diag(img) | higher threshold on the length of each side |
| roi_angle_th_low | 20 | deg | lower threshold on the internal angles |
| roi_angle_th_high | 160 | deg | higher threshold on the internal angles |
| ROI Matching (cf. Subsection III-C) | | | |
| roi_bb_padding | 2 | px | padding value of bounding boxes |
| roi_h_side | 80 | px | side value of a ROI after homography |
| roi_hamming_th | 2 | - | maximum value of hamming distance to detect a marker |
| Trajectory Planning (cf. Section IV) | | | |
| braccio_params(1) | 71 | mm | distance between ground and joint 1 (base) |
| braccio_params(2) | 125 | mm | distance between joint 1 (base) and joint 2 (shoulder) |
| braccio_params(3) | 125 | mm | distance between joint 2 (shoulder) and joint 3 (elbow) |
| braccio_params(4) | 195 | mm | distance between joint 3 (elbow) and the end-effector tip |
| braccio_params(5) | 0 | mm | 'a' (aka 'r') DH parameter for joint 5 (roll) |
| z_min | $-5$ | mm | minimum z-value of target points, in robot frame |
| joint_safety_radius(1) | 0 | mm | safety distance from ground for the joint 1 |
| joint_safety_radius(2) | 30 | mm | safety distance from ground for the joint 2 |
| joint_safety_radius(3) | 30 | mm | safety distance from ground for the joint 3 |
| joint_safety_radius(4) | 30 | mm | safety distance from ground for the joint 4 |
| joint_safety_radius(5) | 0 | mm | safety distance from ground for the joint 5 |

*Table III. Description of the main parameters of the implementation. Default values were used in the experiments.*