Università Ca' Foscari Venezia

Master Degree in Artificial Intelligence and Data Engineering

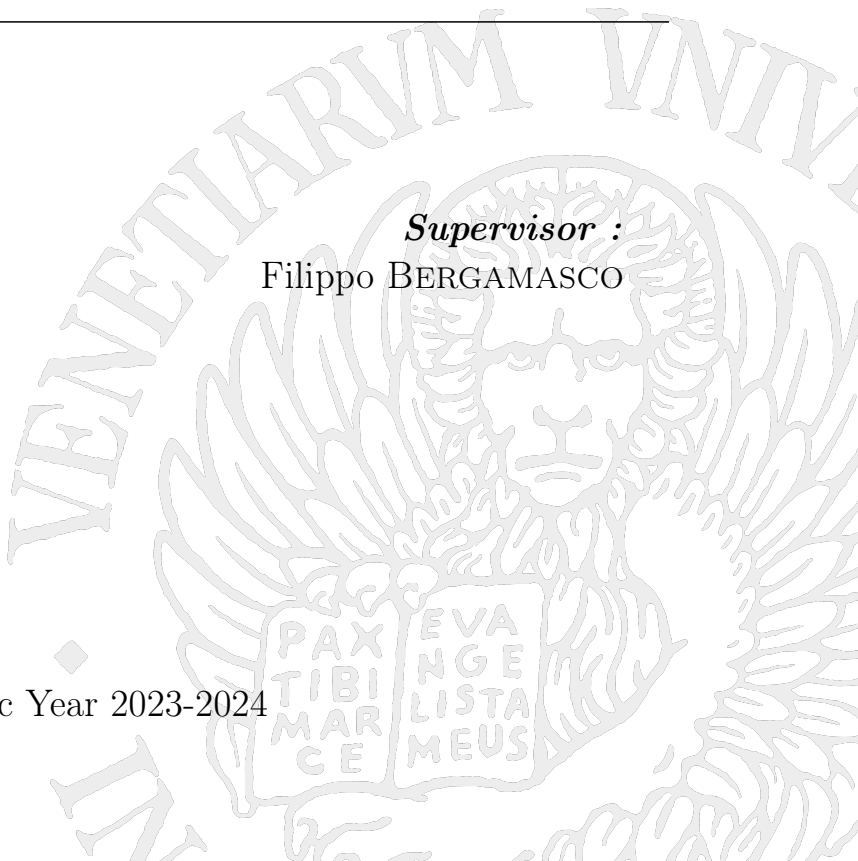
Final Thesis

Object pose estimation using RGB-D camera and robotic manipulator

---

Graduand:
Alessio Narder 868071

Supervisor :
Filippo Bergamasco



Academic Year 2023-2024

Contents

1 Abstract

This work proposes a potential ROS2-based pipeline for object pose estimation, using a robotic manipulator and an RGB-D camera. The proposed pipeline includes a set of coordinated nodes that address key sub-problems: camera intrinsic calibration, hand-eye calibration, point cloud stitching, object detection, and object pose estimation. A detailed and comprehensive description of the solution design and implementation is provided, along with insights into common issues encountered during the testing phase. Finally, experimental results are presented to evaluate the system's effectiveness, and potential improvements are discussed.

2 Introduction

Object pose estimation is the problem of estimating the position and orientation of an object in a given space. This is a very well-known problem in both robotics and computer vision and the solution complexity can vary, depending on the involved environmental factors and adopted hardware, as well as tool pose constraints and physics properties of the materials. To solve this problem, a system able to explore the space in which the tools are placed and a sensor capable of detecting the tools are both required. Even though, in some cases, the need to explore the space dynamically can be relaxed, the ability to move the sensors can lead to a more precise and informative view of the surrounding environment, which can be a crucial capability in some scenarios. A robotic manipulator is an excellent solution when dealing with operations in 3D-space. A robotic manipulator is a robotic arm whose end-effector offers the possibility to carry a tool, which can then be used to perform automation tasks or inspection operations. A robotic manipulator can be built as a 6-degree-of-freedom system made by a set of linked joints with hydraulic or electric actuators. This design guarantees high flexibility and reachability. An example of robotic manipulators is given in the image 1, where some Universal Robot manipulators are shown

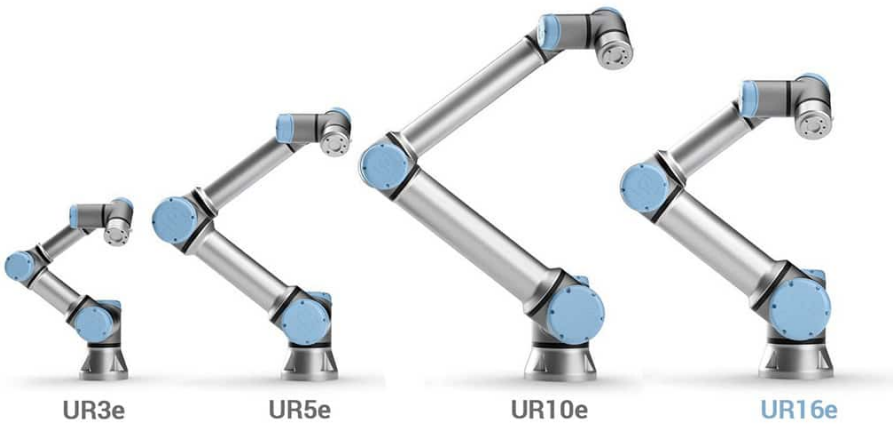


Figure 1: 6-dof robotic manipulators

Even though robotic manipulators have been in the industry and research institutes since the 70's, their usage has always been limited to automation tasks, where the operations to perform were programmed offline with limited to absent feedback and low-level programming languages. In recent times, thanks to the increasing computational power and more advanced connection protocols, the programming paradigm has shifted to a more high-level and online fashion, where a computer is used to control the machine with a high-level programming language, such as C++ or Python, along with sensors to provide feedback. For these reasons, it can be easily understood why a robotic manipulator can be used to explore a given space.
When dealing with detection, on the other hand, the best-fit sensor heavily depends on

Figure 2: The Intel Realsense D435i RGB-D camera

the specific use case. In some scenarios, where depth information must be retrieved online or/and with high confidence, lasers or equivalent scanners can be used. These tools, while very precise, often lack relevant information associated with the colors or other visual properties of the environment, leading sometimes to a not sufficiently rich representation of the space. One class of sensors able to preserve the visual properties along with depth information is the RGB-D cameras. This type of sensor is capable of streaming images as well as depth frames in real-time, performing sensor fusion and frame synchronization on the integrated chip. One example of RGB-D camera is the Intel Realsense D435i camera, shown in the image 2, which uses stereoscopic cameras and RGB camera to stream both depth and RGB frames.

RGB-D cameras offer a good tradeoff in terms of depth accuracy and image resolution and the onboard frame synchronization guarantees consistent data without any further intervention.

3 Background

This work aims to define a pipeline that uses an RGB-D camera and a robotic arm to detect and localize a set of known tools with constrained poses in space. To build such a pipeline, we first need to solve a set of well-known sub-problems, covering both robotics and computer vision. In particular, we need to clarify the way a robotic manipulator moves in space, in terms of coordinate systems and geometric transformations, i.e. its kinematic model. Moreover, we should find a way to describe the geometrical properties of an optical camera in terms of distortion and intrinsic parameters, as well as be able to understand where the camera is positioned with respect to the robot coordinate system using a rigid transformation. In the following chapters, the mathematical models and theoretical background of each sub problem will be given

3.1 Robotic Manipulator Kinematics

Robot kinematics involves applying geometric principles to study the motion of kinematic chains with multiple degrees of freedom, which make up the structure of robotic systems [1][2]. This branch of robotics accounts for the study of the position and its higher-order derivatives, such as velocity, acceleration, jerk and many more. The robot kinematics is expressed in terms of links, i.e. the elements of the kinematic chain. The links are numbered starting from the base of the arm, which is fixed and is numbered as link 0. The first moving link is link 1, and so on, until the last link, which is link n

3.1.1 Links

One way to think of a manipulator is as a chain made up of joints connecting a number of bodies called links. The robot may move in space thanks to these joints, which link nearby links and permit relative motions. The link between two bodies when their relative motion comprises sliding surfaces is referred to as a lower pair. Depending on whether or not it has loops, the joint chain can be either open or closed. Although there are many other types of lower pairs, as shown in Figure 3, revolute joints are the most common form found in most industrial manipulators.

The immobile arm base, sometimes referred to as link 0, is where the links are numbered from. Link 1 is the first moving body, and so on, all the way to link n, the free end of the arm. A minimum of six joints are needed to place an end-effector generally in 3D-space. This makes sense if we consider that three parameters are required to describe a point's position and orientation in three dimensions: three for the position and three for the orientation. A link can be characterized physically by a variety of factors, including its weight, inertia, material strength and stiffness, and more. However, a link is only taken into consideration as a rigid body that specifies the interaction between two adjacent joint axes of a manipulator to determine the kinematic equations of the mechanism. Lines in space define joint axes. A line in space, or vector direction, about which link i spins in relation to link i_1 defines joint axis i. It turns out that two integers can be used to specify a connection for kinematic reasons. These numbers specify the relative positions of the two axes in space. There is a common joint axis connecting adjacent links. The distance between one link and the next along this shared axis is one interconnection parameter. This parameter is often referred to as the **link offset**. The offset of joint axis i is conventionally called d_i. The second parameter describes the angle of rotation between one

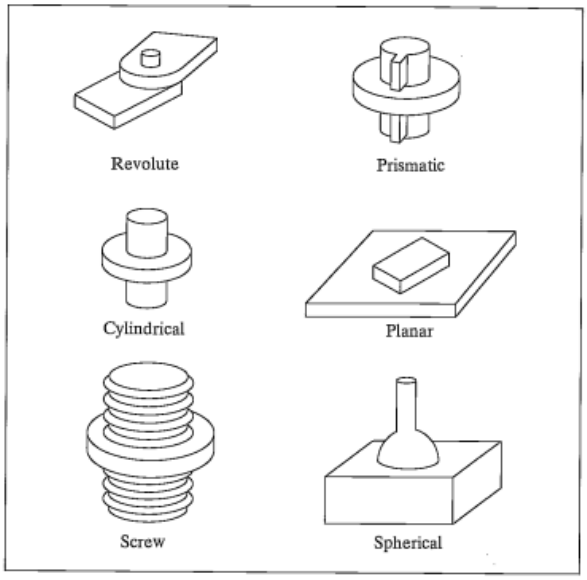


Figure 3: The **six types** of lower-pair joints. Image from [3]

link and its neighbor along a common axis. This is called the joint angle θ_i

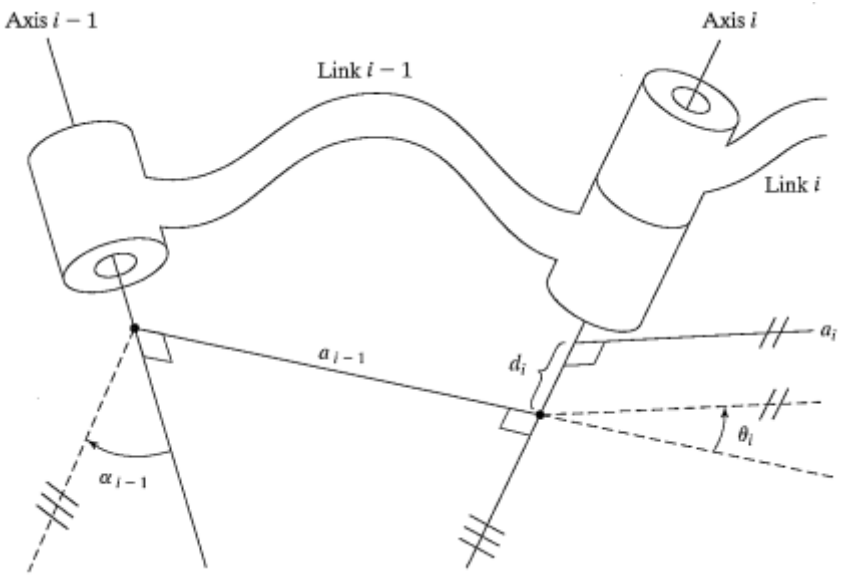


Figure 4: The link offset, d, and the joint angle, θ, are two parameters that can be used to characterize the relationship between adjacent link. Image from [3]

3.1.2 Denavit-Hartenberg convention

Hence, any robot can be described kinematically by giving the values of four quantities for each link. Two describe the link itself, and two describe the link's connection to a neighboring link. In the usual case of a revolute joint, is called the joint variable, and the other three quantities would be fixed link parameters. The definition of mechanisms by means of these quantities is a convention usually called the Denavit—Hartenberg notation.

In order to represent a chain of links using this notation, one must follow the following procedure:

1. **Define the coordinates frame**
   The z-axis of each frame must be aligned with the axis of rotation. The x-axis is chosen in a way that is perpendicular to the z-axis and the y-axis direction is derived by applying the right-handed convention. The origin of the frame is located at the point where the z-axes of two consecutive joints intersect or where the z-axis intersects the x-axis of the previous frame

2. **Define the four-link parameters**
   For each link, we need to compute the four parameters that describe the transformation from the previous link with respect to the frame origin previously computed.

   - θ: The angle needed to rotate about the z-axis to align the x-axis of the previous frame with the x-axis of the current frame
   - d: The distance to move along the z-axis to reach the origin of the current frame
   - a: The distance to move along the x-axis to the point where the z-axis of the current frame begins
   - α: The angle to rotate around the x-axis to align the z-axis of the current frame with the z-axis of the next frame

In the following image 5, the Universal Robots UR20 link description is given along with its Denavit-Hartenber parameters, as reported in the official documentation [4]

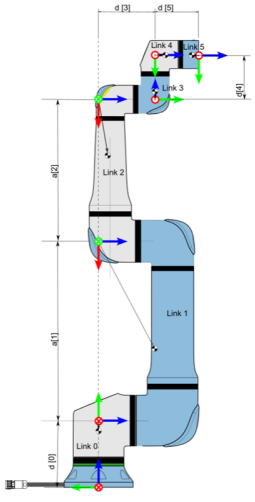


Figure 5: Universal Robots UR20 chain of links description

3.1.3 Manipulator kinematics

Now we want to construct the transform that defines frame $\{i\}$ relative to the frame $\{i-1\}$. This transformation, usually, is a function of the four-link parameters. But, for any given robot, this transformation will be just a function of only one variable. The

Joint	θ [rad]	a [m]	d [m]	α [rad]
1	0	0	0.2363	$\pi/2$
2	0	-0.8620	0	0
3	0	-0.7287	0	0
4	0	0	0.2010	$\pi/2$
5	0	0	0.1593	$-\pi/2$
6	0	0	0.1543	0

Table 1: Denavit-Hartenberg Parameters for UR20

other three parameters, in fact, are fixed by mechanical constraints, meaning that they cannot change. From this observation, we are able to reformulate the kinematics problem into n subproblems of defining a frame for each link. In order to solve these subproblems, namely ${}^{i-1}_{i}T$, we need to reformulate each subproblem into four smaller subproblems. *Each of these four transformations will be a function of one link parameter only and will be simple enough that we can write down its form by inspection.* We begin by defining three intermediate frames for each link—$\{P\}$, $\{Q\}$, and $\{R\}$.

The figure 6 shows the same pair of joints as before with frames $\{P\}$, $\{Q\}$, and $\{R\}$ defined. Note that only the $\hat{X}$ and $\hat{Z}$ axes are shown for each frame, to make the drawing clearer. Frame $\{R\}$ differs from frame $\{i-1\}$ only by a rotation of α_{i-1}.

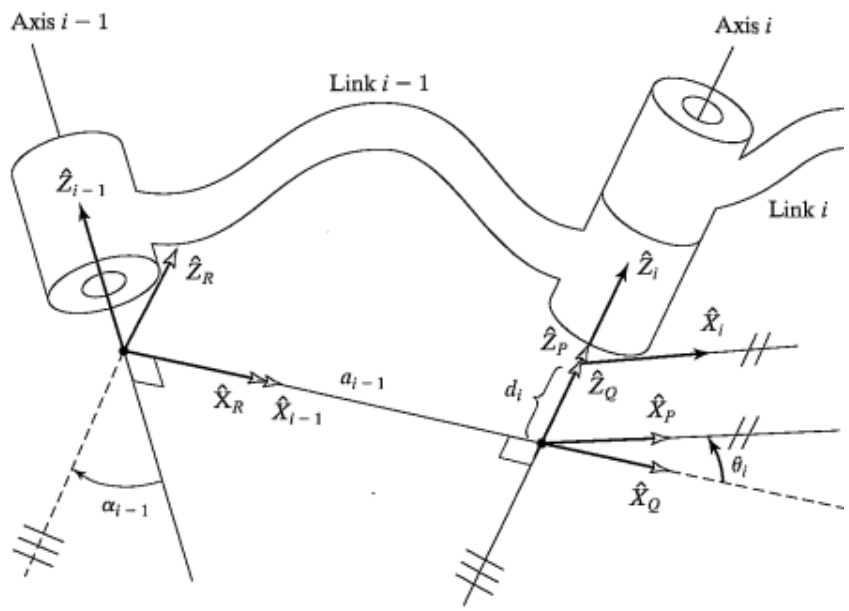


Figure 6: Location of intermediate frames P, Q, and R. Image from [3]

Frame {Q} is related to frame {R} by a translation a_{i-1}. Frame {P} differs from {Q} by a rotation θ_i, and frame {i} differs from {P} by a translation d_i. To express the transformation that converts vectors defined in frame {i} to their representation in frame {i-1}, we can write:

$$^{i-1}P = ^{i-1}\ T_R^Q T_Q^P T_P^i P \tag{3.1}$$

or

$$^{i-1}P = {}^{i-1}_{i}T \; {}^{i}P \tag{3.2}$$

where

$${}^{i-1}_{i}T = {}^{i-1}T^{Q}_{R}T^{P}_{Q}T^{i}_{P} \tag{3.3}$$

Considering these transformations, we see that previous equation may be written as

$${}^{i-1}_{i}T = R_X(a_{i-1})D_X(a_{i-1})R_Z(\theta_i)D_Z(d_i) \tag{3.4}$$

or

$${}^{i-1}_{i}T = \text{Screw}_X(a_{i-1}, \alpha_{i-1})\text{Screw}_Z(d_i, \theta_i) \tag{3.5}$$

where the notation $\text{Screw}(\hat{Q}, r, \phi)$ represents a combined operation of a translation along the axis $\hat{Q}$ by a distance r and a rotation about the same axis by an angle ϕ. Expanding equation (3.4), we obtain the general form of the transformation from frame i to frame $i-1$:

$${}^{i-1}_{i}T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1}d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{3.6}$$

That is, we have obtained the homogeneous transformation from one joint to another. We can also obtain the inverse transformation ${}^{i}_{i-1}T$ by just applying the inverse homogeneous transformation:

$${}^{i}_{i-1}T = \begin{bmatrix} R^1 & -R^{-1}T \\ 0 & 1 \end{bmatrix} \tag{3.7}$$

Then, we can multiply the link transformations to find the single transformation that relates frame N to frame 0:

$${}^{0}_{n}T = {}^{0}_{1}T \; {}^{1}_{2}T \; {}^{2}_{3}T \; ... \; {}^{n-1}_{n}T \tag{3.8}$$

3.2 The Pinhole Camera Model

Let's consider the central projection of points in space onto a plane. Let the center of projection be the origin of a Euclidean coordinate system, and consider the plane Z = f, which is called the image plane or focal plane. Under the **pinhole camera model**, a point in space with coordinates X = (X, Y, Z)T is mapped to the point on the image plane where a line joining the point X to the center of projection meets the image plane.

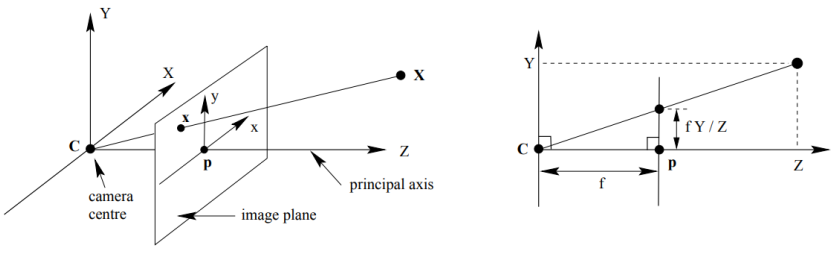


Figure 7: **C** is the camera center and p the principal point. The image plane is located in front of the the camera center. Image from [5]

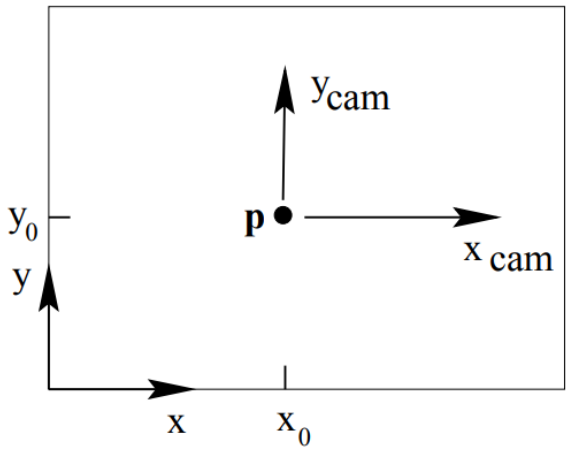


Figure 8: Image and camera coordinate systems. Image from [5]

Using triangular similarity, we can state that the point $(X, Y, Z)^T$ is mapped to the point $\left(\frac{fX}{Z}, \frac{fY}{Z}, f\right)^T$ on the image plane. From this observation, we can say that

$$(X, Y, Z)^T \mapsto \left(\frac{fX}{Z}, \frac{fY}{Z}\right)^T \tag{3.8}$$

describes the central projection mapping from world to image coordinates. This is a mapping from the euclideans 3D-space $\mathbb{R}^3$ to 2D-space $\mathbb{R}^2$.

The center of this projection is referred to as the *camera center*, also known as the *optical center*. The line extending from the camera center and perpendicular to the image plane is known as the *principal axis* or *principal ray* of the camera. The intersection of the principal axis with the image plane is termed the *principal point*. The plane that passes through the camera center and is parallel to the image plane is called the *principal plane* of the camera.

3.2.1 Central projection

We can express the central projection as the mapping of the world points and image points using homogeneous coordinates. Using this method, we can rewrite (3.8) as:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.9}$$

We can now use the notation X for the world point represented by the homogeneous 4-vector $(X, Y, Z, 1)^T$, x for the image point represented by a homogeneous 3-vector, and P for the 3×4 homogeneous *camera projection matrix*. Then (3.9) can be rewritten in a more compact form

$$x = PX$$

3.2.2 Camera Calibration Matrix

The expression (3.9) assumed that the origin of coordinates in the image plane is at the principal point. However, this could not be always the case. In particular, we have a mapping:

$$(X, Y, Z)^T \mapsto \left(\frac{fX}{Z} + p_x, \frac{fY}{Z} + p_y \right)^T$$

where $(p_x, p_y)^T$ represents the coordinates of the principal point. This equation can be reformulated in homogeneous coordinates as

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{pmatrix} = \begin{pmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}. \tag{3.10}$$

Now, writing

$$K = \begin{pmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.11}$$

we obtain (3.11) in the concise form

$$x = K[I \mid 0]X_{\text{cam}}. \tag{3.12}$$

This matrix K is referred to as the *camera calibration matrix*. In (3.12) we have written $(X, Y, Z, 1)^T$ as X_{cam} to make evident that the camera is assumed to be located at the origin of a Euclidean coordinate system having the principal axis of the camera pointing straight down the Z-axis, and the point X_{cam} is expressed in this coordinate system. Such a coordinate system may be called the *camera coordinate frame* and is a standard frame representation.

3.2.3 Camera rotation and translation

We can express points in space in terms of a specific coordinate frame, known as the *world coordinate frame*. The two coordinate frames are obtained via a rotation and a translation, as shown in figure 9, in this specific order. If $\mathbf{X}$ is a non-homogeneous 3D-vector representing the coordinates of a point in the world coordinate frame, and $\mathbf{X}_{\text{cam}}$ represents the same point in the camera coordinate frame, then we can write

$$\mathbf{X}_{\text{cam}} = R(\mathbf{X} - \mathbf{C}),$$

where $\mathbf{C}$ represents the coordinates of the camera center in the world coordinate frame, and R is a 3×3 rotation matrix representing the orientation of the camera coordinate frame.

This equation may be written in homogeneous coordinates as

$$\mathbf{X}_{\text{cam}} = \begin{pmatrix} R & -R\mathbf{C} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} R & -R\mathbf{C} \\ 0 & 1 \end{pmatrix} \mathbf{X}. \tag{3.12}$$

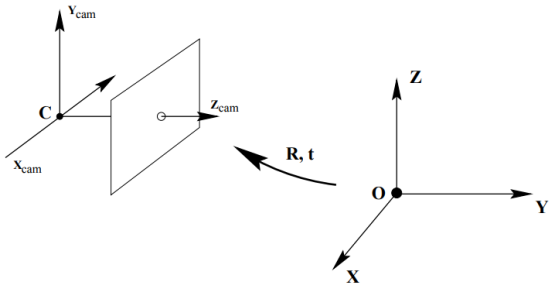


Figure 9: The Euclidean transformation between the world and camera coordinate frames. Image from [5]

Putting this together with (3.11) leads to the formula

$$\mathbf{x} = KR[I \mid -\mathbf{C}]\mathbf{X}, \tag{3.12}$$

where $\mathbf{X}$ is now expressed as a point in the world coordinate frame. This is the mapping that we obtain using a pinhole camera. A general pinhole camera, $P = KR[I \mid -\mathbf{C}]$, has 9 degrees of freedom: 3 for K (the elements f, p_x, p_y), 3 for R, and 3 for $\mathbf{C}$. The parameters in K are called the *intrinsic camera parameters*, or the *internal orientation* of the camera. The parameters of R and $\mathbf{C}$, instead, describe the camera roto-translation with respect to the world reference frame. These parameters are called the *extrinsic parameters* or the *exterior orientation*.

It is often convenient not to make the camera center explicit, and instead to represent the world to image transformation as $\mathbf{X}_{\text{cam}} = R\mathbf{X} + \mathbf{t}$. In this case, the camera matrix is simply

$$P = K[R \mid \mathbf{t}], \tag{3.13}$$

where from (3.12) $\mathbf{t} = -R\mathbf{C}$.

3.3 Camera Calibration

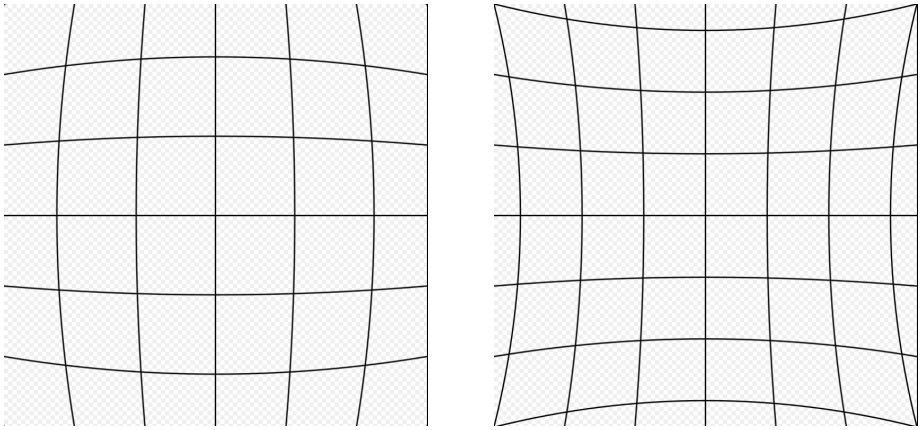
As explained in the previous chapter, the pinhole camera model is described by the *camera calibration matrix*, whose parameters map 3D points in the camera coordinate frame to the image plane (camera pixel frame). While these parameters are sufficient for an ideal pinhole camera, real-world cameras often introduce distortion, causing points to appear closer or farther from the camera than they are. To address this issue, distortion models are needed to describer the position of distorted points relatively to the central point. In this chapter, we will discuss two of the most common distortion models and review calibration procedures used to extract both distortion parameters and the camera matrix for real-world pinhole cameras.

3.3.1 Types of Distorsions

In geometric optics, rectilinear projection means that straight lines are projected as straight lines also in the image. With distortion, we deviate from this type of projection, generating an optic aberration. The most relevant types of distortion are two: radial distortion and tangential distortion. Radial distortion means that straight lines appear as curved lines and arise from the spherical shape of the lenses, which causes light to be

refracted differently between its center and edges [6]. Tangential distortion, on the other hand, means that some areas in the image look nearer than expected and occurs because the image-taking lens is not aligned perfectly parallel to the imaging plane.

- **Radial Distortion**
  Radial distortion can appear in two different fashions: barrel distortion and pincushion distortion. With barrel distortion, the image zoom decreases with distance from the optical axis, as shown in the image 10a. This type of distortion is common on lenses with concave shape



(a) Barrel distortion (b) Pincussion distortion

Figure 10: Two different types of radial distortion

  On the other hand, with pincushion distortion the image zoom increases with the distance from the optical axis, as shown in the image 10b. Convex spherical lenses tend to have pincushion distortion. In both cases, the distortion is quadratic, meaning they increase as the square of distance from the center. The radial distortion can be described with the following model:

$$x_{distorted} = x(1 + k_1 r^2 + k2_r^4 + k_3 r^6)$$

$$y_{distorted} = y(1 + k_1 r^2 + k2_r^4 + k_3 r^6)$$

- **Tangential distortion**

  In tangential distortion, some areas in the image may look nearer or farther than expected, as shown in the image 11. Tangential distortion occurs when the image-taking lens is not aligned perfectly parallel to the imaging plane

  The tangential distortion can be described with the following model:

$$x_{distorted} = x + [(2p_1 xy + p_2(r^2 + 2x^2))]$$

$$y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

In short, to express distortion, we need 5 parameters, known as distortion coefficients, given by: $(k_1\ k_2\ p_1\ p_2\ k_3)$

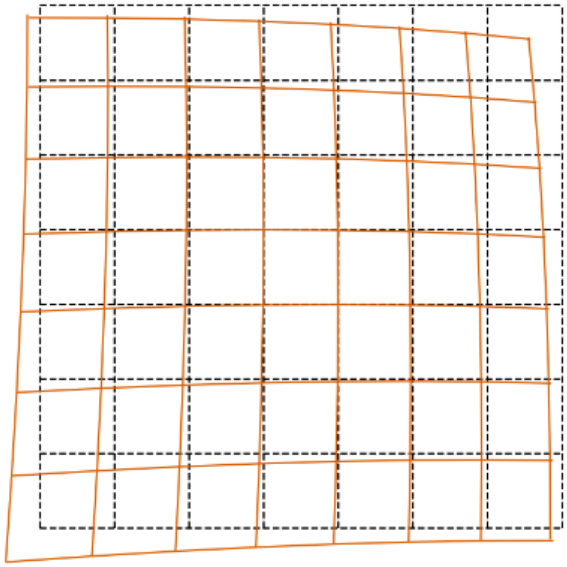


Figure 11: Tangential distortion

3.3.2 Calibration method

Camera calibration requires finding both the calibration matrix K and the distortion parameters. Even though camera calibration calibration be performed using different strategies depending on the specific use case and camera properties, one common algorithm used in practice to solve this problem is the standard camera calibration algorithm offered by OpenCV [7], which is mostly based on Zhang's algorithm [8] and on the [9]. This algorithm requires the knowledge of the position of some keypoints in the world space, the so-called *objectpoints*, and the knowledge of the keypoint positions in the projected camera frame, the so-called image *imagepoints*, over a set of samples. The algorithm then iteratively updates the calibration matrix parameters and the distortion coefficients by comparing the reprojection of the *objectpoints* in the image using the estimated values and real ones, that is, the original image points

To provide a more in-depth understanding of the procedure, a pseudo-code representation of the algorithm is given below:

Algorithm 1: Camera Calibration Algorithm (Zhang's Method)

Input: A set of N 3D points $\mathbf{X_i}$ (object points) and their corresponding 2D projections $\mathbf{x_i}$ (image points) across multiple images

Output: Intrinsic parameters matrix K, rotation matrices R_i, translation vectors t_i, distortion coefficients k_1, k_2, k_3, p_1, p_2

1 **Initialization**

2 Estimate the homographies H_i between 3D object points and 2D image points for each image using a planar object, like a checkerboard.

3 **Estimate Intrinsic Parameters**

4 Using the homographies H_i, solve the following equation to estimate the intrinsic matrix K:

$$v_{ij} = \begin{bmatrix} h_{i1}h_{j1} & h_{i1}h_{j2} + h_{i2}h_{j1} & h_{i2}h_{j2} \end{bmatrix}^T$$

Minimize the following system to find K:

$$v_{12}(K) = 0, \quad v_{11}(K) - v_{22}(K) = 0$$

where $v_{ij}(K)$ is a vector computed from the homography and K is the intrinsic parameter matrix:

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

5 **Estimate Extrinsic Parameters**

6 For each image, calculate the rotation R_i and translation t_i using the homography H_i and the estimated K.

7 **Refine Parameters (Levenberg-Marquardt Optimization)**

8 Minimize the reprojection error between the observed image points $\mathbf{x_i}$ and the projected 3D points $\hat{\mathbf{x}}_i$:

$$E = \sum_i \|\mathbf{x_i} - \hat{\mathbf{x}}_i\|^2$$

where $\hat{\mathbf{x}}_i = K[R_i|t_i]\mathbf{X_i}$. This step optimizes all parameters: intrinsic, extrinsic, and distortion coefficients (radial and tangential).

9 **Output the Calibration Results**

10 Return the optimized values for K, R_i, t_i, and distortion coefficients.

The Zhang's method allows to recover a plausible of K, while the Levenberg-Marquardt based reprojection improves the accuracy of the whole parameters set. The algorithm repeats this procedure until convergence or a given termination criteria is met.

3.4 Hand-Eye Calibration

The hand-eye calibration problem requires estimating the transformation that links a tool (eye) with the end-effector (hand). This problem is in practice more complex than the camera calibration problem because it involves both computer vision and rigid transformation, making the acquisition of the algorithm inputs more difficult. In this chapter, the hand-eye calibration will involve the estimation of the homogeneous transformation of a camera with respect to the flange, i.e. the last movable part of the robot. In order to correctly state the problem, we need to introduce 4 relevant poses involved in the

calibration process:

- **Flange-Base Pose of the Robot**
  This is the fundamental pose of a robot as it represents its end-effector relative to the fixed base on which it moves. Generally, this pose is provided through the robot's control software, supplied by the robot's manufacturer.

- **Checkerboard-Camera Pose**
  In the case of a camera, it is possible to estimate the position of a calibration tool, obtaining the pose of the checkerboard relative to the camera's optical center, or vice versa. This estimation is achieved through standard calibration techniques that are easy to implement, involving the recognition of a specific calibration object, usually a checkerboard, whose geometry is known in advance.

- **Camera-Flange Pose**
  Obtaining this pose is equivalent to solving the Hand-Eye Calibration problem. Knowing the pose of the camera relative to the flange allows us to easily express the fourth and final pose of the robot-camera system.

- **Checkerboard-Base Pose of the Robot**
  This pose is a composition of the three previous transformations. Knowing the three previous transformations allows the system to express, in coordinates relative to the robot's reference system, poses relative to the camera.

The goal of the hand-eye calibration is to estimate the camera-flange pose by using a set of checkerboard-camera and flange-base poses. In the image 12, a simple representation of the experimental setup needed to solve the problem is given:

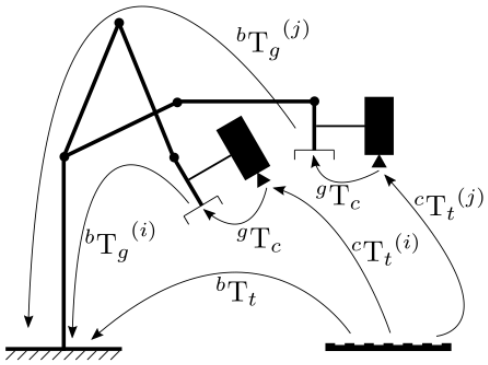


Figure 12: Hand eye calibration

By using a calibration object, commonly a checkerboard, whose geometric characteristics are known in advance, it is possible to move the robot, and with it the camera, to capture multiple images of the calibration object from different positions.

From these images, by analyzing the distortion of the checkerboard caused by the perspective transformation, the pose of the camera relative to the calibration object can be extracted. This phase is called the extraction of extrinsic parameters, which are commonly presented in the form of a 4x4 rotation-translation matrix:

$$\begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$$

where R represents a 3x3 rotation matrix and t a three-dimensional translation vector.

The position of the robot relative to its reference system, on the other hand, is directly retrieved from the robot itself, depending on the specific model of the robot, relying on the manufacturer's specifications and its software environment.

This set of information allows for the classical formulation of the Hand-Eye Calibration problem, in the form of a system of homogeneous equations:

$$AX = XB$$

In the classical formulation, considering two positions 1 and 2 of the camera-robot system, matrix A represents the rotation-translation matrix resulting from the transformation between the two poses of the sensor, calculated as:

$$A = A_2 A_1^{-1}$$

Similarly, matrix B represents the transformation between the flange poses, calculated as:

$$B = B_2^{-1} B_1$$

Finally, X represents the solution to the problem, that is, the transformation between the optical center of the sensor and the flange.

3.4.1 Calibration methods

Now that the hand-eye calibration problem has been explained, we can proceed by looking over some algorithms used in practice to solve the problem. Even though there exist many different algorithms can be used to solve this problem, in this work we will present two algorithms that represent two different approaches: the first one, which solves the problem in two steps (separable solutions), and the second one, which solves the problem in a single step (simultaneous solutions). The first presented algorithm is the one proposed by Tsai and Lenz [10], which proposes a solution to the problem $AX = XB$ which involves solving the system of equations by decomposing the problem into its rotational and translational components, solving them sequentially. Then, Horaud and Dornaika [11] propose a different formulation of the problem, equivalent to the previous one, also introducing quaternions and exploiting their properties to propose two new solution approaches that can be used in both formulations: one, similar to the previous method, solves the rotational and translational parts separately; the other, more efficient, solves for rotation and translation simultaneously. As for the camera calibration, the presented algorithms are adopted by OpenCV and their performances will be tested in the experimental part of this work. A brief explanation will be given for each of the presented algorithms, along with a pseudo-code representation and some additional considerations.

- **Separable solutions approach**

  In the solution proposed by Tsai and Lenz [10], we start from the setup shown in figure 12 and proceed by solving the problem for each station in terms of rotation,

and then in terms of translations. Then, we solve a system of equations using the least square methods, to obtain the hand-eye transformation. The solving algorithm is based on representing a transformation as a combination of a rotation by an angle θ around an axis passing through the origin with direction cosines $[n_1, n_2, n_3]$, followed by a translation T, represented as follows:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R \begin{bmatrix} x \\ y \\ z \end{bmatrix} + T$$

or equivalently:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

where (x, y, z) and (x', y', z') are the coordinates of any point before and after undergoing the transformation.

R is a 3x3 orthonormal matrix of first type ($\det(R) = 1$):

$$R = \begin{bmatrix} n_1^2 + (1 - n_1^2)\cos\theta & n_1 n_2(1 - \cos\theta) - n_3 \sin\theta & n_1 n_3(1 - \cos\theta) + n_2 \sin\theta \\ n_1 n_2(1 - \cos\theta) + n_3 \sin\theta & n_2^2 + (1 - n_2^2)\cos\theta & n_2 n_3(1 - \cos\theta) - n_1 \sin\theta \\ n_1 n_3(1 - \cos\theta) - n_2 \sin\theta & n_2 n_3(1 - \cos\theta) + n_1 \sin\theta & n_3^2 + (1 - n_3^2)\cos\theta \end{bmatrix}$$

Having R as a unitary eigenvalue, and $P_R \equiv [n_1 n_2 n_3]^T$ as the rotation axis, the rotation matrix R can be expressed using its eigenvector, which consists of vector P_R scaled by a function of θ, using a modified version of the well-known Rodrigues' formula:

$$P_r = 2 \sin\frac{\theta}{2} \begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix} \quad \text{with} \quad 0 \leq \theta \leq \pi$$

Thus, it is possible to express the rotation R as a function of P_r:

$$R = \left(1 - \frac{|P_r|^2}{2}\right) I + \frac{1}{2}(P_r P_r^T + \alpha \cdot \text{Skew}(P_r))$$

where $\alpha = \sqrt{4 - |P_r|^2}$ and $\text{Skew}(P_r)$ is the skew-symmetric matrix trace defined as:

$$\text{Skew}(V) = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

Algorithm 2: Tsai and Lenz Algorithm

Data: Transformations P_{gij}, P_{cij}
Result: Values of P'_{cg}, P_{cg}, and T_{cg}

for *each pair of stations* i, j **do**
  Construct the system of equations for P'_{cg} as the unknown:

$$\text{Skew}(P_{gij} + P_{cij}) \cdot P'_{cg} = P_{cij} - P_{gij}$$

  where:
  - P_{cij} represents the transformation between the two camera stations.
  - P_{gij} represents the transformation between the two flange stations.

  Solve for P'_{cg} using least squares.

Calculate the rotation angle:

$$\theta_{Rcg} = 2\tan^{-1}\left(\left|P'_{cg}\right|\right)$$

Calculate P_{cg}:

$$P_{cg} = \frac{2P'_{cg}}{\sqrt{1 + P'_{cg} \cdot P'^{T}_{cg}}}$$

where P_{cg} represents the rotational part of the transformation from the camera to the flange.
Finally, calculate the translational component T_{cg}:
Construct the system:

$$(R_{gij} - 1)T_{cg} = R_{cg}T_{cij} - T_{gij}$$

Solve for T_{cg} using least squares.

Even though this method is efficient and computationally fast, the authors in the paper have highlighted that some precautions must be taken in order to obtain good calibration results. In particular, given that the algorithm first solves the problem in terms of rotations and then in terms of translations, it is necessary to collect pose data with a wide variety of rotations at each station and significant differences in station positions, while keeping the distance from the calibration object low and constant among the poses. Another algorithm that solve the problem in two steps is the one proposed by F. Park, B. Martin Robot [12], which solves $AX = XB$ on the euclidean group.

- **Simultaneous solution approach**

The problem $AX = XB$ has a solution X which represents the hand-eye homogeneous transformation between two different camera poses A and the relative robot poses B. Another way to approach this problem, as proposed by Radu Horaud and Fadi Dornaika in [11], is to solve the equation $MY = M'YB$, where M and M' represent the camera's 3x4 perspective matrices from different positions and Y is

the transformation being solved. In the reference paper, they show that the two formulations are equivalent. For this reason, they proposed two formulations of the algorithm: the first one, which solves the problem first in terms of rotations and then in terms of translations (separable solutions), and an optimized one, which solves rotations and translations simultaneously.

Algorithm 3: Horaud and Dornaika Algorithm

Data: Given transformations A, B, and unknown X.

Result: Determine the transformation matrix X (rotation and translation between hand and eye).

1 **for** *Each camera position i* **do**

2 Construct the system:

$$M_2 Y = M_1 Y B_{ij}$$

Solve for Y, the transformation between the hand frame and the calibration frame.

- Use unit quaternions to solve for rotation:

$$v' = q * v * q$$

Solve the closed-form solution using eigenvalue decomposition of matrix A.

- Use least-squares for translation estimation after determining rotation.

Optimization: For simultaneous optimization of rotation and translation:

- Use the Levenberg-Marquardt method for minimizing the error function:

$$f(q,t) = \sum \|v' - q * v * q\|^2 + \sum \|q * p * q - (K - I)t - p' * q\|^2$$

Minimize using non-linear optimization to obtain the most robust solution.

In their work, Radu Horaud and Fadi Dornaika found that the optimized version of their algorithm outperforms the two steps version. This is explained by inevitable error propagation from the first step (solve for rotations) and the second step (solve for translations), which is removed in the simultaneous version. Another example of an algorithm that simultaneously solves the problem is the one proposed by K. Daniilidis [13], using Dual Quaternions, also implemented in OpenCV

3.5 3D Reconstruction and Stereo vision

3D reconstruction is the process of creating a three-dimensional model of a scene or object from multiple two-dimensional images or other data sources. Even though this problem can be attacked using very different techniques, in this chapter we will focus on the 3D reconstruction based on epipolar geometry, which offers a geometrical approach to the problem. In particular, we are interested in explaining how it is possible to interpolate the position of a point in world coordinates using two-dimensional images coming from two different image sensors placed in different positions. This is also the method used by every animal equipped with stereo-vision (or binocular vision), which enables the

perception of depth in the surrounding environment through eyes. Moreover, this is the method adopted by many RGB-D camera producers, where two different image sensors are used to generate the so-called pointcloud, i.e. a set of 3d points that describes the space in front of the camera. In the next sub-chapter, a theoric background over the epipolar geometry will be given, along with some common methods that can be used to perform 3D-reconstruction

3.5.1 Epipolar geometry

The epipolar geometry between two views describes the geometric relationship that we can establish between two image planes and the set of planes that intersect along the baseline, which is the line connecting the camera centers. Now, we aim to use epipolar geometry to find out how a 3D point is projected in the two image planes. This information will then be useful to solve another problem: knowing the projection of a point on two different image plane, where is the 3D point located in space?

We start with a point $\mathbf{X}$ in 3D-space. This point is imaged in two views, at $\mathbf{x}$ in the first, and $\mathbf{x}'$ in the second. Now we want to exploit the relation between the corresponding image points $\mathbf{x}$ and $\mathbf{x}'$. As shown in figure 13 the image points $\mathbf{x}$ and $\mathbf{x}'$, space point $\mathbf{X}$, and camera centers are coplanar. We can note this plane as π. Consequently, the rays back-projected from $\mathbf{x}$ and $\mathbf{x}'$ intersect at $\mathbf{X}$, and they are also coplanar, lying in π.

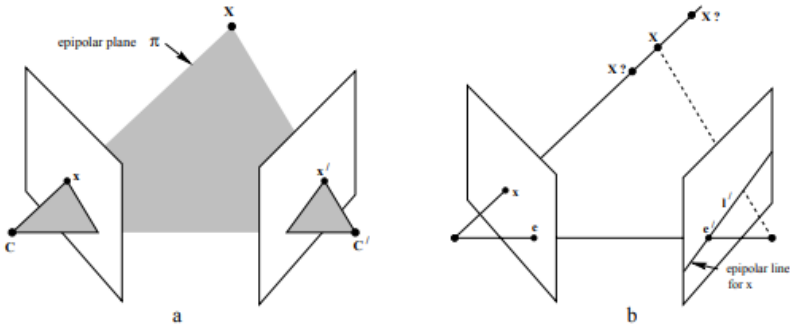


Figure 13: Point correspondence geometry. In the left image, the two cameras are represented by their centers, C and C', along with their image planes. The camera centers, a point X in 3D space, and its corresponding image points x and x' all lie within the same plane π. On the right, an image point x back-projects to a ray in 3D space, defined by the first camera center C and the point X. This ray is projected as a line l' in the second view. Since point X, which projects to x, must lie on this ray, its corresponding image in the second view must lie on l'. Image from [5]

We can proceed by supposing to know $\mathbf{x}$, and then finding out how the corresponding point $\mathbf{x}'$ is constrained. The plane π is formed by the baseline and the ray defined by $\mathbf{x}$. We already know that the ray associated with the point $\mathbf{x}'$ is contained in π, so the point $\mathbf{x}'$ lies on the line of intersection l' of π with the second image plane. This line l' is the image in the second view of the ray back-projected from $\mathbf{x}$, i.e. the *epipolar line* corresponding to $\mathbf{x}$. This information is particularly important when we need to find the corresponding point of $\mathbf{x}$, because we can avoid looking over the complete image, but only on the line l'

Now that we have explained the geometric constraints between $\mathbf{x}$ and $\mathbf{x}'$ 14, we can proceed by introducing three fundamental entities of epipolar geometry:

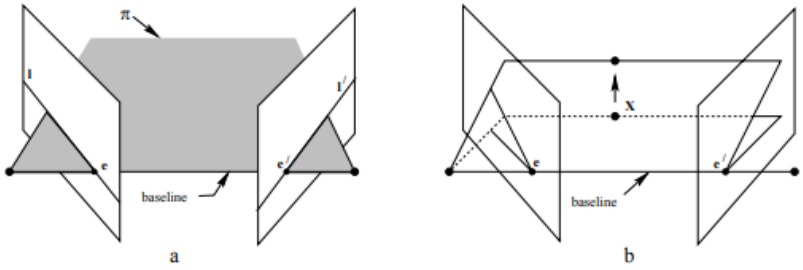


Figure 14: Epipolar geometry. (a) The camera baseline intersects each image plane at the epipoles e and e'. Any plane π that contains the baseline is called an epipolar plane and intersects the image planes along corresponding epipolar lines l and l'. (b) When the position of X changes, the epipolar planes rotates accordingly around the baseline. All epipolar lines converge at the epipole. Image from [5]

- **Epipole**

  This is the point of intersection of the line joining the camera centers with the image plane

- **Epipolar plane**

  The is a plane containing the baseline

- **Epipolar line** Represents the intersection of an epipolar plane with the image plane. All epipolar lines intersect in one point, the epipole.

3.5.2 Fundamental Matrix

In this section one relevant matrix will be presented: the *fundamental matrix*. The properties of this matrix will play a significant role in the next charapter, where we will explain how we can recover the position of a point in space by using its projections over two different camera view. The *fundamental matrix* can be thought of as the algebraic representation of the epipolar geometry. In particular, it represents algebraically two relevant informations:

- For each point x in one image, we can find an epipolar line l' in the other image.
- For any given point x' in the second image matching the point x lies in the epipolar line l'

The fundamental matrix can be obtained using two methods: a geometric method or an algebraic method. We will now explain how to derive the matrix using both methods.

- **Geometric approach**

  Let π be *any* plane in space and X a 3D point on that plane. The point x and x' are the image of X in the two images respectively. We got:

  – x' must be on the epipolar line l' which corresponds to the image of the ray passing through x and X.

– An homography H_π exists mapping each x_i to its corresponding point x'_i through π.

For a given point x', the epipolar line l' is the line that passes through both x' and the epipole e':

$$l' = e' \times x'$$

$$l' = [e']_\times x'$$

where

$$[e']_\times = \begin{pmatrix} 0 & -e'_z & e'_y \\ e'_z & 0 & -e'_x \\ -e'_y & e'_x & 0 \end{pmatrix}$$

Since $l' = [e']_\times x'$ and $x' = H_\pi x$, we have the following:

$$l' = [e']_\times H_\pi x = Fx$$

The fundamental matrix F can be defined as:

$$F = [e']_\times H_\pi$$

where H_π represents the mapping from one image to the other through any plane π. Since $[e']_\times$ has rank 2, F is a matrix of rank 2.

- **Algebraic approach**

  The ray that is back-projected from x by the first camera is expressed as:

  $$X(\lambda) = P^+ x + \lambda C$$

  Here, P^+ represents the pseudo-inverse of the projection matrix for the first camera, and C is the null-vector satisfying:

  $$PC = 0$$

  The ray that is back-projected from x by the first camera is given by:

  $$X(\lambda) = P^+ x + \lambda C$$

  In particular, the points $X(0) = P^+ x$ and $X(\infty) = C$.

  When these points are projected onto the second image using P', they yield the epipolar line:

  $$l' = (P'C) \times (P'P^+ x)$$

  The point $P'C$ represents the projection of the first camera's center into the second image, which is the epipole e'.

  Thus, we can express:

  $$l' = [e']_\times (P'P^+)x = Fx$$

  Where $F = [e']_\times (P'P^+)$ and $(P'P^+) = H_\pi$, which provides the explicit form of the homography for point transfer between the two image planes.

The fundamental matrix satisfies one important condition: for any pair of correspondent points $x \leftrightarrow x'$ in the two images, the following equation is true:

$$x'^T F x = 0$$

If the correspondence $x \leftrightarrow x'$ holds, then we can say that x' lies on the epipolar line $l' = Fx$, and thus:

$$0 = x'^T l' = x'^T F x$$

This is significant because **the fundamental matrix can be computed from point correspondences alone**, without needing to know the projection matrices. Moreover, the point correspondences can be obtained not only from different static camera views but also from the motion of a single camera over a static environment (by pure motion).

3.5.3 Triangulation

Triangulation is the process of determining the 3D position of a point X, given its images x and x' in two views. The most intuitive approach, that is performing "back-projection" of the rays from x and x', does not give accurate results in real-world scenarios, because back-projected rays don't intersect generally. To solve this problem, some estimation methods must be used. In this paragraph, a brief presentation of three of the most common estimation methods will be given.

- **Mid-point based**

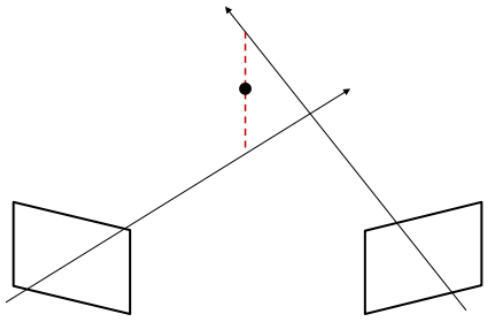


Figure 15: Mid-point based triangulation. Image from [14]

  Choose the midpoint of the common perpendicular between the two rays. Despite its simplicity, this method typically does not yield optimal results due to various approximations (especially when the two cameras are not equidistant from the mid-point).

- **Using linear-methods**
  This method is the most commonly used, offering a good balance between simplicity and accuracy.

  Consider the projection of a point X:

$$\mathbf{u} = w \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = PX$$

Each component of $\mathbf{u}$ can be written as:

$$wu = \mathbf{p}_1^T X, \quad wv = \mathbf{p}_2^T X, \quad w = \mathbf{p}_3^T X$$

By substituting w into the first two equations and considering two projections $\mathbf{u}, \mathbf{u}'$, we obtain a system of four equations in the form $AX = 0$:

$$u\mathbf{p}_3^T X = \mathbf{p}_1^T X$$

$$v\mathbf{p}_3^T X = \mathbf{p}_2^T X$$

$$u'\mathbf{p}_3'^T X = \mathbf{p}_1'^T X$$

$$v'\mathbf{p}_3'^T X = \mathbf{p}_2'^T X$$

There are two approaches to solving the system:

1. Using **homogeneous coordinates**, compute the vector X that minimizes $\|AX\|$ subject to $\|X\| = 1$. The solution is the unit vector which corresponds to the smallest singular value.
2. By setting $X = (x, y, z, 1)^T$, the system $AX = 0$ reduces to a set of four **non-homogeneous** equations in three unknowns, which can be solved using **Linear Least Squares**.

Even though this can be a limit situation, the second method has an intrinsic problem. The system-based approach, in fact, assumes that the points are not at infinity. If this happens, the algorithm will suffer of numerical instability

- **Using iterative linear methods**

  The issue with linear methods is that $\|AX\|$ has no geometric interpretation.

  For each equation in the form:

  $$u\mathbf{p}_3^T X = \mathbf{p}_1^T X$$

  We obtain an error:

  $$\epsilon = u\mathbf{p}_3^T X - \mathbf{p}_1^T X$$

  However, the geometric error is instead given by:

  $$\epsilon' = u - \frac{\mathbf{p}_1^T X}{\mathbf{p}_3^T X}$$

  The error ϵ' can be obtained by scaling the error ϵ by the factor:

  $$\frac{1}{w} = \frac{1}{\mathbf{p}_3^T X}$$

  Problem: The weights depend on the 3D point X, which needs to be estimated.

  Solution: Iteratively estimate X and update the weights, starting with an initial weight of 1.

Algorithm 4: Iterative Weight Update Algorithm

Initialize: Set $w = 1$, $w' = 1$;

while *not converged* **do**

Step 1: Solve the following equations:

$$\frac{1}{w} u \mathbf{p}_3^T X = \mathbf{p}_1^T X \frac{1}{w}$$
$$\frac{1}{w} v \mathbf{p}_3^T X = \mathbf{p}_2^T X \frac{1}{w}$$
$$\frac{1}{w'} u' \mathbf{p}_3'^T X = \mathbf{p}_1'^T X \frac{1}{w}$$
$$\frac{1}{w'} v' \mathbf{p}_3'^T X = \mathbf{p}_2'^T X \frac{1}{w}$$

Step 2: Update the weights:

$$w = \mathbf{p}_3^T X, \quad w' = \mathbf{p}_3'^T X$$

end

3.6 ROS - Robot Operating System

ROS is an open-source set of libraries and a framework released in November 2017 that can be used to develop robotic applications. It allows developers to build an application as a set of independent nodes able to communicate using a specific pub-sub protocol. Its latest revision, ROS2, it offers a powerful QoS configuration model and a DDS based communication system. In the next subchapter, a brief overview of the ROS2 framework architecture will be given, as well as some relevant information about the main interfaces offered by the system.

3.6.1 System architecture

ROS2 framework is built upon a layered architecture. Each layer provides a different level of abstraction and has a specific role. A brief description of the system architecture is given in the schema 16

- **Application Layer and Client interface**

  This layer corresponds to the developer *workspace*. The *workspace* offers the possibility to develop independent routines, called `nodes`, which can communicate with each other using the ROS2 IDL and perform calculations. The app `nodes` can be orchestrated using a so-called `launch file`, which is a python script whose role is to perform the startup procedure and enforce runtime policies. Nodes can be created using the ROS2 CLI and written from scratch in different languages (C++, Python, Node.js, and others). ROS2 nodes can also be installed on the machine via `apt-get` or using the ROS2 package manager, `rosdep`. In order to interact with the ROS2 API, each node can use the appropriate client library. The officially supported client libraries are `rclcpp` and `rclpy`, for C++ and Python respectively.

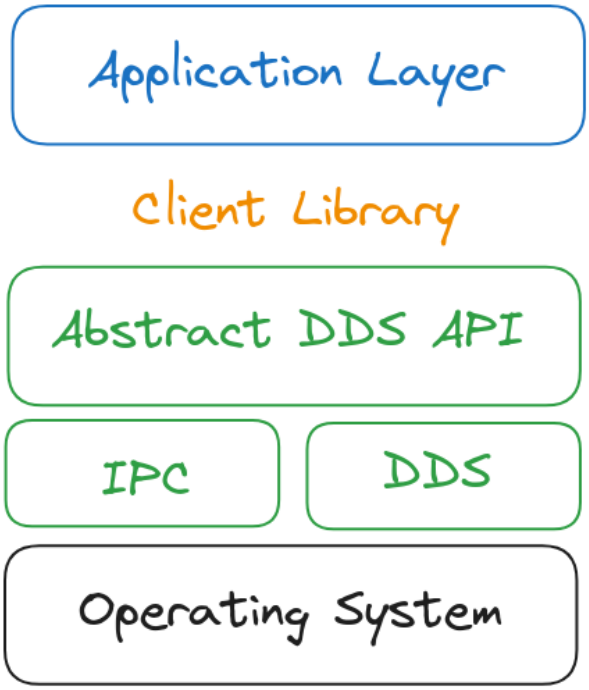


Figure 16: ROS2 system architecture

- **DDS abstract interface**

  ROS2 uses a communication mechanism based on DDS (Data Distribution Service), an industry-standard protocol used for real-time applications. The DDS abstract interface masks the DDS provider-specific implementation and offers a set of standard communication API and QoS settings that decouples the application layer from the API offered by the DDS provider. Moreover, the DDS interface exposes communication primitives, such as topics, services and actions and a cross-language IDL (Interface Definition Language) which can be used to transmit typed and structured data, such as primitive types (float, booleans, integers and more) and lists

- **DDS providers and IPC**

  This is the communication layer, responsible for reliable and real-time packet transport and delivery. The standard DDS provider offered in ROS2, since the first ROS2 LTS (Long Term Support Release) version, is the eProxima FastDDS, but other providers can be used. In ROS2 the nodes' discovery is performed using multicast advertising, making ROS2 nodes able to communicate also with nodes running on a different machine, if visible inside the same network interface. Other than the DDS based communication, ROS2 uses IPC communication (Intra Process Communication) via shared memory, making nodes hosted in the same machine able to communicate with very low latency.

3.6.2 Main Interfaces

As previously written in the DDS abstract interface, ROS2 offers a set of different communication primitives that can be used to adopt different communication patterns, both synchronous and asynchronous. In the following list, the most relevant primitives will be

described: topics, services and actions. Along with them, also the role of parameters will be briefly explained.

- **Topics**:
  - **Use case**: Publish-subscribe messaging for continuous data streams.
  - **Description**: Topics are used for unidirectional, many-to-many communication where a *publisher* node sends messages, and one or more *subscriber* nodes receive them. It is suited for real-time data exchange, such as sensor readings or robot movement commands.
  - **Example**: A robot's camera publishes images on a topic, and multiple nodes subscribe to process the images.
- **Services**:
  - **Use case**: Synchronous, request-response interactions.
  - **Description**: Services are used when one node needs to send a request and wait for a response from another node. It's a two-way communication where a *client* sends a request, and a *server* responds.
  - **Example**: A node requests a path planning service to compute a route, and the service returns the result.
- **Actions**:
  - **Use case**: Long-running, goal-oriented tasks.
  - **Description**: Actions are used for operations that take time to complete, allowing for feedback and cancellation. A node can send a goal to an *action server*, which can provide periodic feedback, report success, or be canceled.
  - **Example**: A robot navigation node sends a goal to reach a specific location, and it receives feedback on the progress while the goal is being executed.
- **Parameters**:
  - **Use case**: Configuration settings.
  - **Description**: Parameters are used to manage node-specific configuration values, such as tunable settings (e.g., threshold values, constants). They can be set or updated dynamically at runtime.
  - **Example**: A robot's speed or sensor update rate can be controlled using parameters.

4 Solution Design

The goal of this work is to present a software system that can estimate the position of some given tools, given that the objects are placed with constrained poses and the space where the objects are positioned can be explored using an RGB-D camera and a robotic arm. In this chapter, we will present the problem to solve more formally, enumerating the functional and non-functional requirements that we need to satisfy to consider a solution acceptable. We will begin with the requirements (functional and non-functional) that will define the software's capabilities. Then, we will proceed to detail the hardware requirements. In the end, a high-level view of the proposed pipeline will be given.

4.1 Functional requirements

Functional requirements define the desired input and outputs of the program. In this case, we aim to build a pipeline that can calibrate the adopted hardware stack and then proceed with the tool pose estimation. For this reason, will will enumerate the required sub-problems to solve, and then, for each subproblem, we will express the desired input and outputs.

1. **Data Gathering**

   The system must be able to collect the needed data from itself. That is, for each problem, the inputs must be taken autonomously

2. **Camera calibration**

   The camera calibration procedure's goal is to estimate the intrinsic parameters of the RGB camera.

   - **Input**: a finite stream of RGB frames. In the stream, images could contain the calibration object, but its detection is not guaranteed
   - **Output**: the camera calibration matrix K

   Given a stream of RGB-frames with a calibration object, estimate the camera intrinsic parameters

3. **Hand-eye calibration**

   The hand-eye calibration procedure requires to estimate the homogeneous transformation that links the hand frame (the flange of the robot) with the camera frame.

   - **Prerequisites**: the camera intrinsics
   - **Input**: A finite and synchronized stream of flange poses paired with RGB-frames containing the calibration object.
   - **Output**: The hand-eye homogenous transformation

4. **Object detection**

   Before estimating the position of a set of objects, we must first detect them. That is, we want to decouple the pose estimation from the object detection to provide a more flexible architecture.

- **Prerequisites**: the camera intrinsic, the hand-eye transformation, RGB-Depth transformation
- **Input**: The set of all the tools to detect as 3D meshes and a finite and synchronized stream of RGB-frames, depth-frames and robot poses taken from an environment linear scansion
- **Output**: A stitched point cloud containing the entire scene and the set of detected tools in the scene, represented as partitions of the stitched point cloud. The point cloud coordinates are expressed with respect to the robot base frame

5. **Object pose estimation**

   - **Input**: The set of detected tools 3D meshes, the set of the pointcloud partitions, the finite and synchronized stream of RGB-frames and depth-frames taken from the environment linear scansion
   - **Output**: The refined positions of the tools with respect to the robot base frame

4.2 Non-functional requirements

Other than the function requirements, we are interested also in the non-functional requirements. Non-functional requirements describe how the system should behave when is operating, so they are not strictly dependent on the functional requirements.

- **Portability**

  The system must not be bound to a specific robot or camera brand and it should be able to run on different operating systems

- **Extensibility**

  The system should be sufficiently generic to allow for further integrations, so the system architecture can be better suited for case-specific scenarios (tools with peculiar characteristics, specific environment conditions and more)

- **Real-Time capabilities**

  The data streams needed to solve each problem must be synchronized, so the time drift between data frames coming from different sources (camera and robot) must be negligible.

4.3 Hardware Requirements

The hardware requirements represent the requirements that must be fulfilled to enable the pipeline to run. Usually, hardware requirements only cover the machine requirements but in this case, we need to enforce specific requirements also on the robot and the RGB-D camera, that

4.3.1 Machine requirements

Machine requirements express the minimum specifications that the runtime machine must fulfill in order to make the program run without issues

- A ROS2-compatible environment
- At least 4GB of RAM
- At least 32 GB of available disk space
- A x86 compatible CPU with quad-core support

4.3.2 Robot requirements

Robot requirements express the minimum specifications that the robotic manipulator must need to achieve sufficient accuracy when computing tool positions and camera optical parameters

- Cartesian space movements support
- Manipulator with at least 6 degrees of freedom
- Less than 0.1mm of X,Y,Z axis error on cartesian poses
- Less than 0.1 degrees of W,P,R error on cartesian poses

4.3.3 Camera requirements

Camera requirements express the specifications that the RGB-camera must reach in order to obtain a coherent pixel-depth alignment

- A RGB-D capable camera with undistorted lenses
- A depth error below 3% when distance is below 1m
- A depth error with zero-sum, i.e. there is no systematic over or under-estimation of the depth values

-

4.4 System design

After having clarified which requirements should be satisfied from the system, we can now delve into the system architecture, which represents, from a high level, the core elements involved in the calibration and the data flux. This representation is agnostic from the implementation and will be used as a starting point for the solution implementation, explained in the next chapter.

As shown in the image 17, the system is made as a pipeline of node. The main nodes are 5:

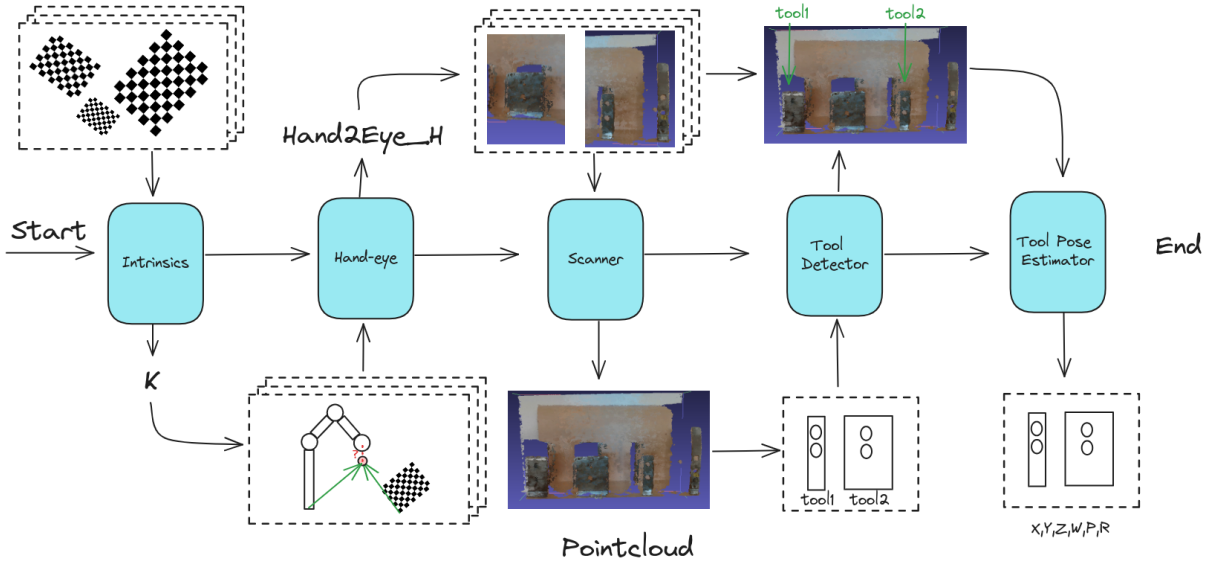


Figure 17: The proposed solution adopts a pipeline architecture, where each node is responsible for providing the needed data for the next node. The output of one node can be used as input from the successive nodes

- **Intrinsics node**

  This node's responsibility is to take care of the entire camera intrinsics calibration procedure. It takes a stream of images containing a calibration pattern and returns K, which is the **camera intrinsics matrix**. Being K crucial for the other procedures, this node represents the first step of the pipeline

- **Hand-eye node**

  This node, as the name suggest, performs the hand-eye calibration. It uses the RGB stream coming from the camera and the poses published by the robot to compute the hand-eye transformation. It also uses K, the camera matrix previously obtained to compute the position of the camera with respect to the calibration object

- **Scanner node**

  This node performs the point cloud stitching, merging a set of partial point clouds with color information. The partial point clouds are obtained using the robot poses, the RGB-frames, the depth frames, the hand-eye transformation, the RGB-depth transformation. It is responsible for both producing the partial point clouds acquired during the scanning and producing a final point cloud, that represents the entire scanned environment.

- **Tool detector node**

  This node uses the stitched point cloud and the tool meshes to perform object detection. The object detection node returns a set of clusters, where each cluster of points is assigned to the most similar tool

- **Tool pose estimatore node**

  This node uses all the previously obtained information to precisely estimate the position of the tools. It represents the last step of the pipeline

5 Solution implementation

The pipelin

5.1 Odometry Node

The odometry node is a ROS2 node responsible for gathering the poses published by the robot and then publishing some relevant odometry information. This node is not strictly required in the pipeline, but it comes in handy to mitigate some synchronization issues encountered during the experimental testing.

5.1.1 Descrition

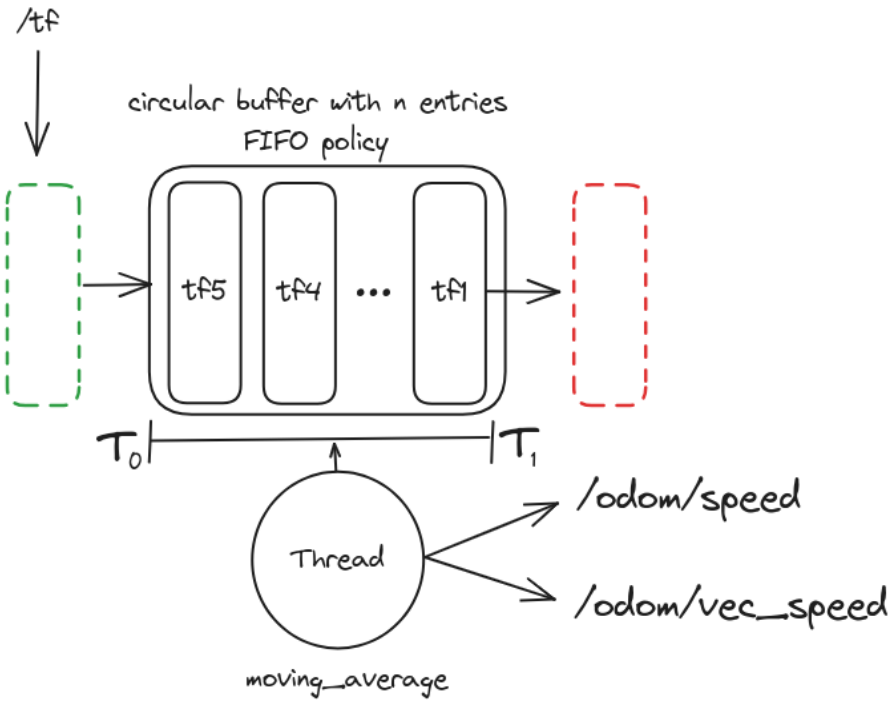


Figure 18: Moving average computation. The queried transforms taken from the /tf buffer are accumulated over a circular buffer, with FIFO policy. A separated thread read the timestamped poses and computes the speed on the axes

The main responsibility of this node is to publish the flange speed. To do so, the node performs the lookup of the pose of the flange with respect to the base in the standard $/tf$ buffer. The poses are then accumulated in a circular buffer of size n, with LIFO (last-in first-out) policy. This policy guarantees that the buffer only contains the last n poses. A separated thread computes the moving average of the speed by inspecting the buffer.

5.1.2 Algorithm

Algorithm 5: Compute the moving average of the speed

Input: The circular buffer of size n
Output: The robot flange speed over x, y and z

```
1 if not buffer.full() then
2 |   return 0,0,0
3 end
4 for i ← 1 to n − 1 do
5 |   δ_s = buffer[i + 1].pos - buffer[i].pos
6 |   δ_t = buffer[i + 1].timestamp - buffer[i].timestamp
7 |   speeds.push(δ_s/δ_t)
8 end
9 return avg(speeds.x, speeds.y, speed.z)
```

5.1.3 Services and topics

This node has no services. The list of topics is provided below

- `/odom/vec_speed`

  This topic publishes the average speed of the flange in each direction component, x, y, z, expressed in m/s

- `/odom/speed`

  This topic publishes the average speed of the flange expressed as the euclidean norm of the x, y, z speeds, expressed in m/s

5.2 Camera Calibration Node

The camera calibration node is a ROS2 node whose goal is to compute the camera's intrinsic parameters. This node is crucial to guarantee good accuracy on the next tasks and is the first node of the previously described pipeline.

5.2.1 State machine

The camera calibration node performs both data gathering and camera intrinsic parameters estimation. A state machine has been used to make the node state bound to a specific task. Figure 19 explains the most relevant states and transitions.

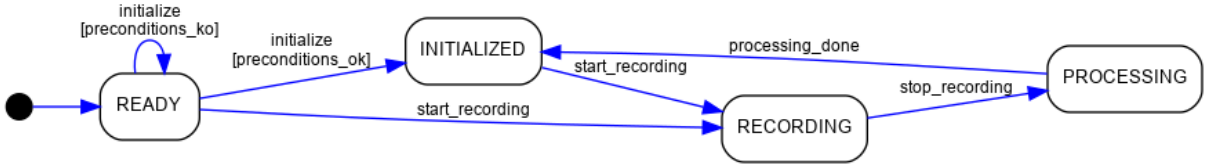


Figure 19: Camera Calibration node state machine. The `ERROR` state is omitted for clarity

The node always begins in the `READY` state. In this state, the state machine can move to the `INITIALIZED` state or remain in the `READY` state, depending on whether the camera matrix has already been computed or not (preconditions are satisfied when the camera matrix is found in the node-specific assets folder). Then, the node can proceed to the `RECORDING` state if the `start_recording` event is triggered. In this state, the node performs the recording task, which will be explained in more detail below. When the recording task is finished due to the `stop_recording` event, the node state moves to the `PROCESSING` state. In this state, the node processes the input data gathered from the `RECORDING` state and proceeds by computing the camera matrix K. When the processing task is done, the `processing_done` event is triggered, moving the node state to the `INITIALIZED` state. For each state, the node can move also on the `ERROR` state, if the event `on_error` is spawned. The node can then restart from the `READY` state if the `on_reset` event is triggered.

5.2.2 Description

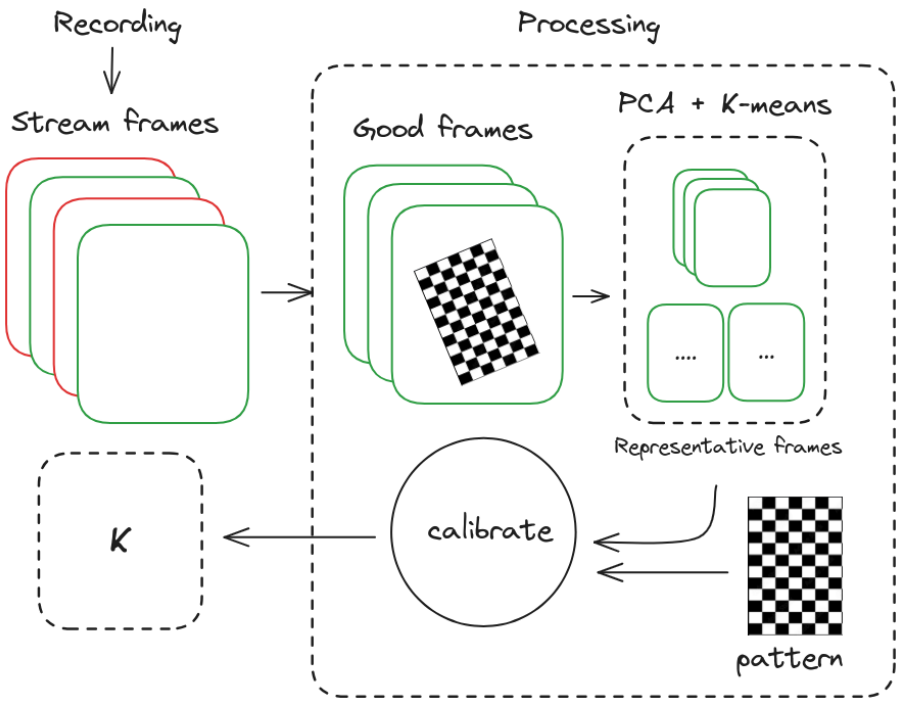


Figure 20: A high level description of the intrinsics node inner working. Frames are collected, filtered and then clustered in a set of representative frames, that will used to recover the camera matrix K

Now that the node's state machine states and transitions have been explained, we can move on to the description of the `RECORDING` and `PROCESSING` tasks.

- **Recording task**

  Recording involves the acquisition of RGB frames coming from the camera. Frames are acquired via ROS2 topic, `/camera/color/image_raw`, and accumulated in a memory buffer. In the phase, no online checks are performed.

- **Processing task**

  The processing task is the task delegated to the computation of the camera matrix K. This task involves several operations. The first operation is data cleaning: the recorded frames are filtered to only consider *good frames*. A frame is *good* if two conditions are satisfied:

  – The frame is not blurred
  – The calibration object is present on the frame

  A frame is considered blurred if the variance of its Laplacian is above a given threshold, specified as a hyperparameter. That is, we want to be sure that the image contains color intensity changes (and we can use the second derivative of the grayscaled images, given by the Laplacian) and then compute the variance of the Laplacian, in order to quantify the changes of intensities in the Laplacian transformed image. Then, if the image is sharp enough, we can proceed by detecting the calibration tool, using OpenCV standard operators (`cv.findChessboardCorners`).

  Then, we proceed with another pre-processing step to extract the top-K representative frames. A frame is considered representative if it effectively represents a set of similar frames. To detect similar frames, we perform K-Means clustering on the PCA-reduced images and consider a frame to be representative of a cluster if it is the closest to the cluster centroid. Selecting representative frames allows to remove the under or over-representation of similar frames in the recorded stream, making the calibration procedure more fair.

  In the end, we can proceed with the calibration procedure, using OpenCV standard operators (`cv.calibrateCamera`), the imagepoints taken from the representative frames, and objectpoints taken from the calibration pattern.

5.2.3 Services and Topics

List of topics

- `/intrinsics/k`

  The estimated camera matrix K, published as a flat array of doubles

- `/intrinsics/corners`

  The detected corners on the image. Is an `Image` type used for debugging and visualization purposes

- `/intrinsics/state`

  The state of the node, intended as the state machine current state, serialized as string

List of Services

- `/intrinsics/start_recording`

  Fires the `start_recording` event, and begins the data gathering

- `/intrinsics/stop_recording`

  Fires the `stop_recording` event and triggers the processing phase

5.3 Hand Eye Calibration Node

The hand-eye calibration node is a ROS2 node, which aims to estimate the hand-eye transformation. This is the second node in the pipeline, and the accuracy of this estimation significantly impacts the error in the scanning phase.

5.3.1 State machine

In the same way as the camera calibration node, the node associates specific operations to states using a state machine and events. The most relevant states and transitions are represented in the image 21.

Figure 21: Hand-eye node state machine

In the same way as the intrinsics node, the node always begins in the `UNINITIALIZED` state. In this state, the state machine can move to the `READY` state or remain in the `UNINITIALIZED` state, depending on whether the dependencies, i.e the camera matrix, can be retrieved or not. If the node state flows to the `READY` state, then the node can proceed to the `RECORDING` state if the `start_recording` event is triggered. In this state, the node performs the recording task, which will be explained in more detail in the next paragraph. When the recording task is finished due to the `stop_recording` event, the node state moves to the `PROCESSING` state. In this state, the node processes the input data gathered from the `RECORDING` state and proceeds by computing the hand-eye transformation using an iterative method. When the processing task is done, the `processing_done` event is triggered, moving the node state to the `INITIALIZED` state. For each state, except for the `UNINITIALIZED` state, the node can move on the `ERROR` state, if the event `on_error` is spawned. The node can then restart from the `READY` state if the `on_reset` event is triggered.

5.3.2 Description

Now that the state machine states and transitions have been explained, we can proceed with the description of the tasks associated with the `RECORDING` and `PROCESSING` states. Moreover, a brief explanation of the dependencies loading task will be given.

- **Dependencies loading**

  In order to obtain an accurate estimation of the hand-eye transformation, we need to recover the camera matrix computed by the camera intrinsic node. To do so, the node, when in the `UNITIALIZED` node and after the `load_dependencies` event, subscribes to the `/intrinsics/k` topic.

- **Recording**

  The recording operation involves the acquisition of data coming from different data sources. The RGB frames are taken via `/camera/camera/color/image_raw` topic

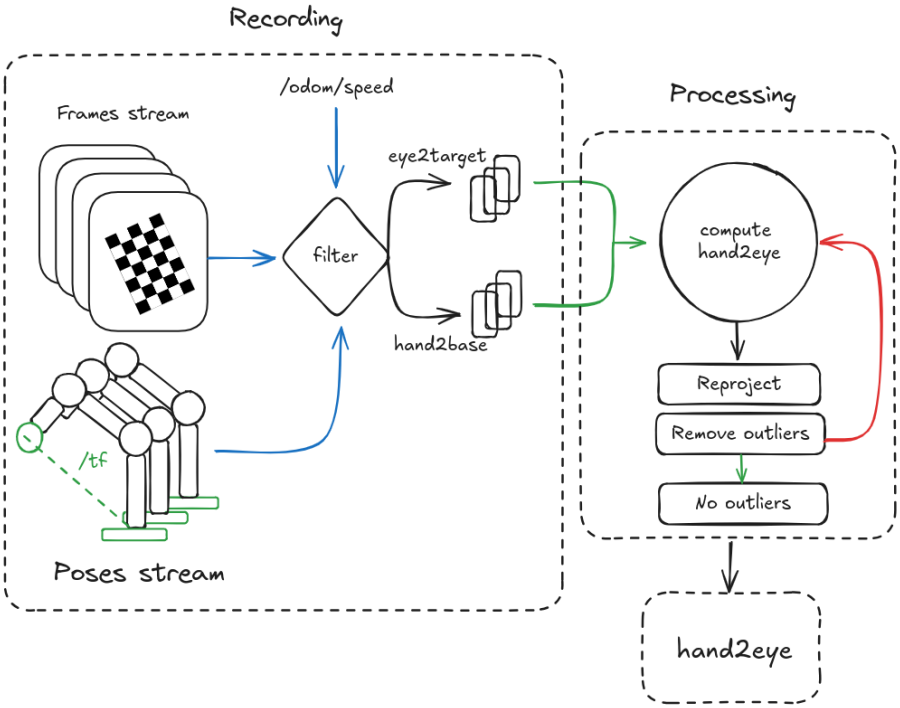


Figure 22: A high-level description of the hand-eye node inner working. Data are taken from different sources and gathered in the recording phase. Recording performs an online filtering process and transforms raw data into two lists, one containing the eye2target transforms and the other containing the hand2base transforms. Processing takes these lists as input and proceeds with estimating the hand-eye with an iterative procedure

subscription, while the flange poses are obtained by querying the transforms buffer using the `tf2_ros.Buffer.lookup_transform` method. The transform lookup buffer is updated asynchronously, while the transform query is performed in the same scope of the RGB frame callback in order to enforce time consistency. Moreover, in the image callback, some relevant filtering operations are performed in order to mitigate the risk of inconsistent or unuseful data acquisition:

- **Image sharpness filter**
  If the RGB frame is not sufficiently sharp (sharpness is computed using the same method of the intrinsic node), skip
- **Calibration object detection**
  If the calibration object is not detected on the image, skip
- **Flange speed filter**
  If the flange speed is higher than a threshold, skip

The filters help reduce the size of the recorded data without affecting the acquisition frequency. Then, in the same callback, the eye2target transform is computed using the OpenCV `cv.solvePnP` method. In this way, the RGB frame callback is able to take the instantaneous flange position (hand2base) as well as the camera pose with respect to the object (eye2target) from consistent data.

- **Processing**

  During the processing phase, the hand-eye node leverages an iterative procedure in order to estimate the hand-eye transform using the data gathered from the recording phase. The hand-eye calibration proceeds by iteratively removing a set of outliers from the re-projected poses of the calibration tool with respect to the robot base frame. A more in-depth description of the procedure is given in the pseudo-code below:

Input: The list of homogeneous transforms `target2eye` `target2eye_hs`
Input: The list of homogeneous transforms `hand2base` `hand2base_hs`
Output: The hand-eye homogeneous transform `eye2hand`

curr_it = 0 ;
while *len(hand2base) > min_samples* ***and*** *curr_it < max_it* **do**
    `eye2hand = cv.calibrateHandEye(target2eye_hs, hand2base_hs)` ;
    `target2base_hs = []` ;
    foreach *hand2base* **do**
        $$T_{\text{target2base}} = T_{\text{hand2base}} \cdot T_{\text{eye2hand}} \cdot T_{\text{target2eye}}$$
        `target2base_hs.append(T_target2base)` ;
    end
    `center = centroid(target2base_hs)` ;
    `dists = euc_distances(target2base_hs, center)` ;
    for $i \leftarrow 0$ **to** *len(dists)* **do**
        if *dist[i] > 2 * std_dev(dists)* **then**
            `remove(target2eye_hs[i])` ;
            `remove(hand2base_hs[i])` ;
        end
    end
    curr_it += 1 ;
end
return *eye2hand*

This iterative approach re-computes the hand-eye transform multiple times, each time with less noisy data, leading to a more accurate hand-eye estimation. This approach makes the hand-eye procedure more robust to the presence of outliers, which can heavily affect the estimated rotation and translation of the camera from the flange. When a termination criteria is met (number of iterations or the number of samples), the hand-eye transform is returned.

5.3.3 Services and Topics

List of services

- `/hand_eye/extrinsics`

  The estimated hand-eye homogenous transformation. The transformation is published as a flat array of doubles

- `/hand_eye/state`

  The state of the node, intended as the state machine current state, serialized as string

List of topics

- `/hand_eye/start_recording`

  Fires the `start_recording event`, and begins the data gathering

- `/hand_eye/stop_recording`

  Fires the `stop_recording event` event and triggers the processing phase

5.4 Scanner Node

The scanner calibration node is a ROS2 node designed to build a point cloud by combining data streams from the RGB camera and the robot. This node is particularly important, being the fourth in the pipeline and the final one that handles both data processing and gathering. The generated point cloud provides feedback on the preceding calibration procedures and completes the data collection necessary for locating tools.

5.4.1 State machine

In the same way as the other node, the node associates specific operations to states using a state machine and events. The most relevant states and transitions are represented in the image 23.

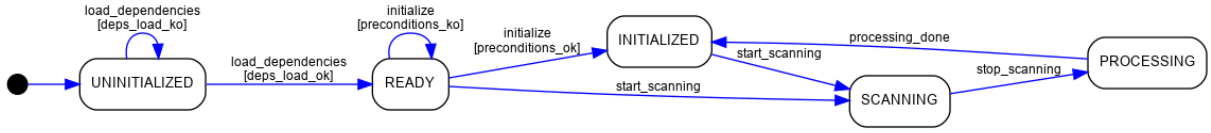


Figure 23: Scanner node state machine. The `ERROR` state is omitted for clarity

The node always begins in the `UNINITIALIZED` state. In this state, the state machine can move to the `READY` state or remain in the `UNINITIALIZED` state, depending on whether the dependencies, i.e the camera matrix and the hand-eye transform, can be retrieved or not. If the node state flows to the `READY` state, then the node can proceed to the `SCANNING` state if the `start_scanning` event is triggered. In this state, the node performs the scanning task, which will be explained in more detail in the next paragraph. When the scanning task is finished due to the `stop_scanning` event, the node state moves to the `PROCESSING` state. In this state, the node processes the input data gathered from the `SCANNING` state and proceeds by computing the overall scene pointcloud. When the processing task is done, the `processing_done` event is triggered, moving the node state to the `INITIALIZED` state. For each state, except for the `UNINITIALIZED` state, the node can move on the `ERROR` state, if the event `on_error` is spawned. The node can then restart from the `READY` state if the `on_reset` event is triggered.

5.4.2 Description

The scanner node requires environmental scanning to acquire the image and depth information needed to build the pointcloud. In this case, we aim to merge data coming from the robot and the camera to build a pointcloud whose entries carry both color information and the position with respect to the robot base frame. In the figure 24, a schematic description of the entire process in given.

Similarly to the hand-eye node, the scanner node acquires data from the robot flange poses using the transforms lookup buffer and the RGB/Depth streams from topics published by the camera, more specifically from the `/camera/camera/color/image_raw`

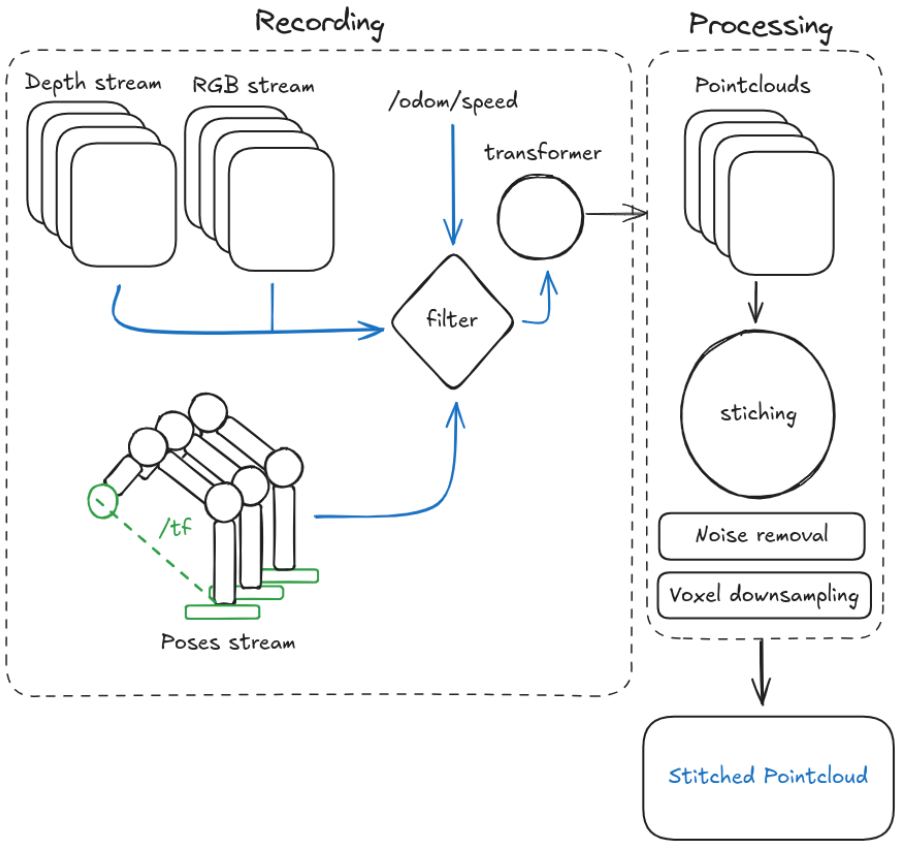


Figure 24: Scanner node high-level description. When the scanner node is in the `RECORDING` phase, it starts accumulating data from both the camera and the robot. Filtering and by pointcloud generation is performed online during the data gathering. In the `PROCESSING` phase, the node performs the pointcloud stitching, followed by some post-processing operations

for the RGB stream and from `/camera/camera/depth/image_rect_raw` for the depth stream. Moreover, informations coming from the `/odom/speed` topic are used in order to perform accurate filtering basing on the robot's flange speed. A more in depth description of the `SCANNING` and `PROCESSING` phase is given below, along with a brief description of the dependencies needed by this node to be in `READY` state

- **Dependencies loading**

  To perform a correct visual alignment of the depth stream with the RGB-stream, we need to load the Depth2RGB transform from the `/camera/extrinsics/depth_to_color` topic and the intrinsics for both the Depth camera and the RGB camera, from the `/camera/depth/camera_info` and the `/intrinsics/k` respectively. The need for both the intrinsics will be explained in the `SCANNING` step. Furthermore, we need to load the hand-eye transformation from the `/hand_eye/extrinsics` topic.

- **Scanning**

  The scanning phase proceeds as follow: from the RGB-image callback, get the latest flange pose, and the latest depth frame. Applies the following filters to the obtained triplet in order to asses data consistency:

1. **RGB frame sharpness filter**
   If the RGB frame is not sufficiently sharp (sharpness is computed using the same method of the intrinsic node), skip
2. **Depth frame timestamp drift filter**
   If the timestamp of the last depth frame is older than a threshold interval, skip
3. **Pose frame drift filter**
   If the timestamp of the last pose is older than a threshold interval, skip
4. **Flange speed filter**
   If the flange speed is higher than a threshold, skip

Then, build a pointcloud from the gathered data using the following procedure, also explained in the figure 25:

1. Using the depth camera matrix, back project the depth pixels in 3D-space
2. Using the Depth2RGB transform, align the backprojected point with respect to the camera frame
3. Using the camera intrinsics, project the 3D-points in RGB camera pixel space to obtain the color informations
4. Transform the 3D-points coordinates from camera frame to robot base frame

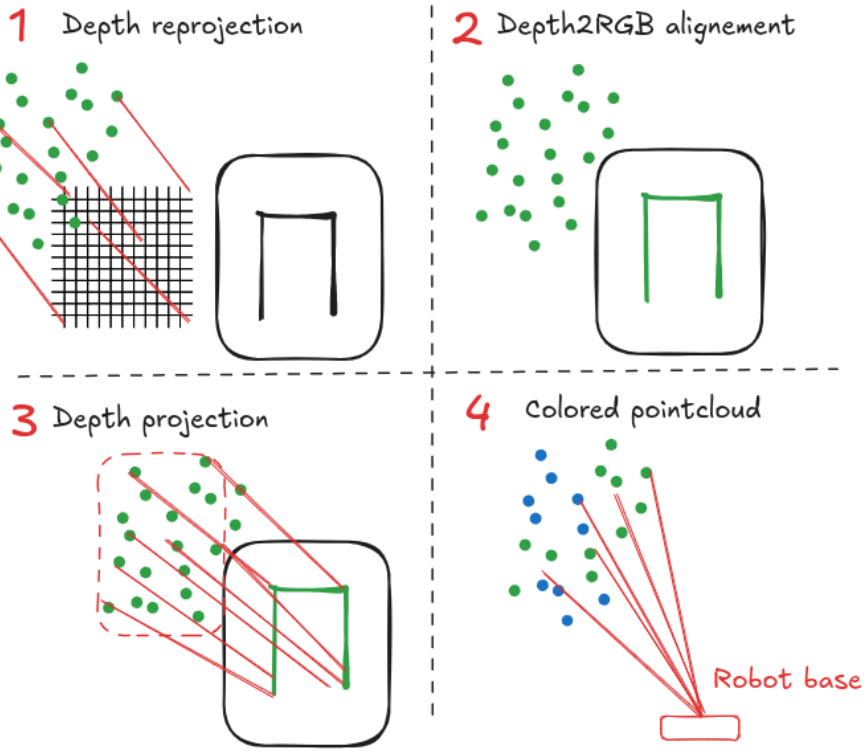


Figure 25: Pointcloud generation from RGB-D streams and robot poses

Then, accumulated the obtained pointclouds in a buffer, that will used to performed the pointcloud stitching

- **Processing**

  In the processing step, all the pointclouds obtained in the previous step are merged into a single pointcloud (pointcloud stiching). Some operations are performed in this phase in order to guarantee a good result

  1. **Pointcloud size reduction**

     For each accumulated pointcloud, only consider a fraction of points contained in a parallelepiped of size $wxhxd$. This ensures that points located far from the pointcloud geometric center are removed. These points' depth, in practice, tend to be interpolated with higher error than the ones nearby the center. This evidence will also be noticed in the Experimental Analysis.

  2. **Noise removal**

     Remove all the noisy voxels applying a 3D Gaussian filter through `open3d.voxel_down_pcd.remove_statistical_outlier`. In particular, remove all the voxels whose distance from the mean is n-times the standard deviation of the distribution of a set of k points.

  3. **Voxel downsampling**

     Applying geometric downsampling: all the voxels within a spere of radius x are compressed into a single central voxel of size x.
     For this task, `open3d.geometry.voxel_down_sample` has been used

  After all these steps, we end up with a stitched and cleaned pointcloud, concluding the processing phase.

5.4.3 Services and topic

- `/scanning/state`

  The state of the node, intended as the state machine current state, serialized as string

- `/scanning/point_cloud`

  The current estimated pointcloud. This topic is active in the `RECORDING` state

- `/scanning/point_cloud_stiched`

  The post-processed pointcloud, published as a `Pointcloud2`

- `/scanning/start_scanning`

  Triggers the `start_scanning` event, which moves the node state to `SCANNING`

- `/scanning/stop_scanning`

  Triggers the `stop_scanning`, which moves the node stat to `PROCESSING`

5.5 Tool Detector Node

The tool detector node is the node responsible for identifying the tools in the stitched pointcloud. The output of this node is a set of labeled clusters taken from the pointcloud, where each label represents the associate tool identifier.

5.5.1 State machine

In the same manner as the previous nodes, the tool detector node is associated to a state machine. In the figure 28, a schematic description of the state machine is given.

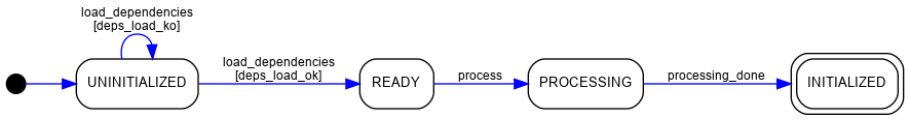


Figure 26: Tool detector state machine

The tool detector node always starts from the `UNINIITALIZED` state. Then, after after the `load_dependencies` event, the node tries to load all the dependencies, i.e. the tools and the stitched pointcloud. The node will move to the `READY` state or the `UNINITIALIZED` state, depending on the ability to load the required dependencies or not. If the node is in the ready state, when the `process` event is spawned, it can move to the processing state. After the termination of the processing phase, noted with the `processing_done` event, it can finally move to the `INITIALIZED` state, which is the final state.

5.5.2 Description

In order to detect tools, we need to identify the set of candidate clusters and then perform matching with the reference tools. The tool detection node accomplishes this task by performing a set of operations, as explained in the figure 27.

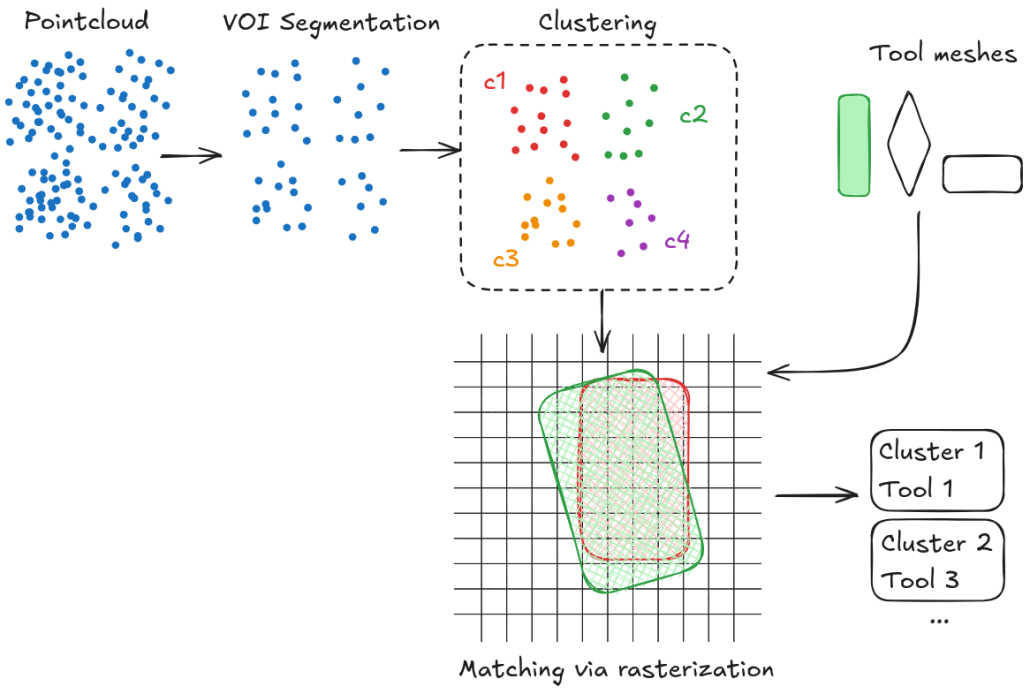


Figure 27: The matching procedure. Clustered voxels and tool meshes are rasterized to compute similarity

The node takes as input the stitched pointcloud, the set of tools to locate as a set of meshes and then it performs a density-based clustering over a volume of interest. Then,

it performs tool matching via rasterization. A more in depth description of the clustering and matching phases is given:

- **Clustering**

  Clustering is performed via `DBScan`, which is a density-based spatial clustering algorithm. In this phase, `DBScan` looks for high-density core samples in the Volume of Interest, and then proceeds to expand the clusters in all the regions where a density criteria is met.

- **Matching**

  Matching is performed via rasterization and 2D-areas overlapping. A brief explanation of the matching algorithm is provided below:

Input: The meshes of tools to locate
Input: The candidate clusters
Output: Labeled clusters

```
1  cluster_labels = [] ;
2  foreach cluster in clusters do
3  |   cluster_sims = [] ;
4  |   foreach mesh in meshes do
5  |   |   rasterized_mesh, rasterized_cluster = rasterize(mesh, cluster) ;
6  |   |   sim = jaccard(rasterized_mesh, rasterized_cluster) ;
7  |   |   cluster_sims.append(sim) ;
8  |   end
9  |   cluster_label = maximum(cluster_sims) ;
10 |   cluster_labels.append(cluster_label) ;
11 end
12 return cluster_labels
```

5.5.3 Services and Topics

- `/detector/detect`

  Trigger the `process` event, which will move the node state to `PROCESSING`

5.6 Tool Pose Estimation Node

The tool pose estimation node is responsible to accurately estimating the position of the detected tools using the information gathered during the pipeline.

5.6.1 State machine

In the same manner as the previous nodes, the tool pose estimation node is associated to a state machine. In the figure 28, a schematic description of the state machine is given.

The tool pose estimation node always starts from the `UNINIITALIZED` state. Then, after after the `load_dependencies` event, the node tries to load all the dependencies, i.e. the clustered labels, the partial pointclouds and the RGB frames. The node will move to the `READY` state or to the `UNINITIALIZED` state, depending on the ability to load the

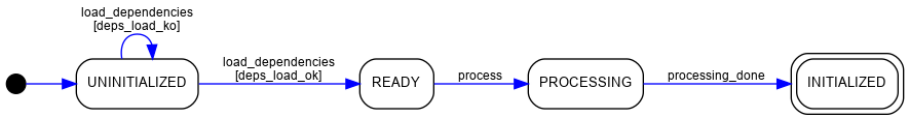


Figure 28: Enter Caption

required dependencies or not. If the node is in the ready state, when the `process` event is spawned, it can move to the processing state. After the termination of the processing phase, noted with the `processing_done` event, it can finally move to the `INITIALIZED` state, which is the final state.

5.6.2 Description

The tool pose estimation node is the final step of the pipeline and allows to recover the pose of the detected tools using both 3D and 2D information. A brief schema of the inner working of this node is given in image 29.

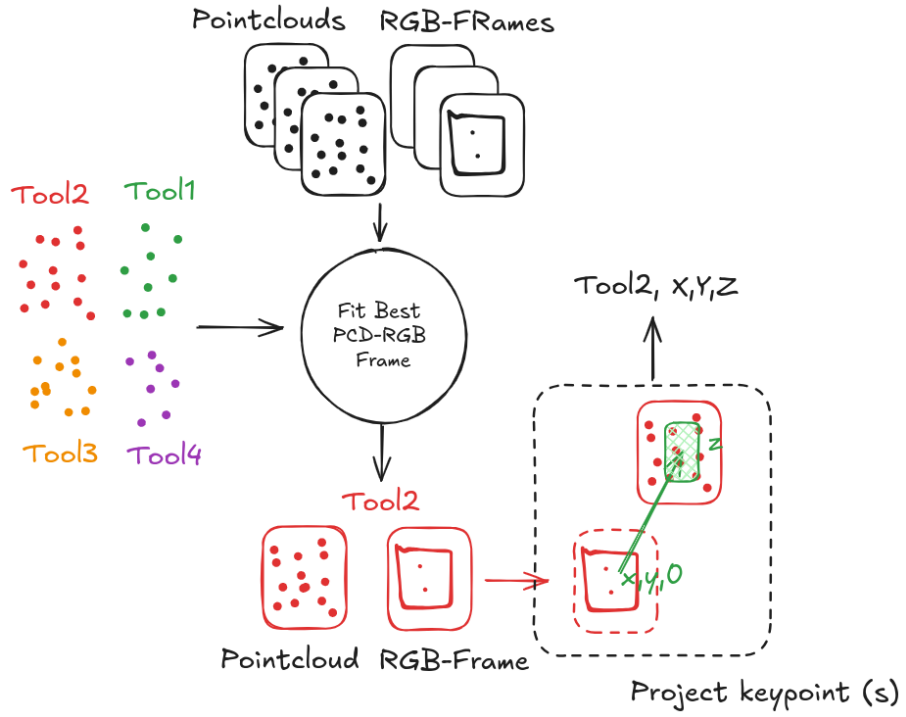


Figure 29: High level description of the tool pose location procedure

The tool pose estimation node takes as input the labeled clusters found in the tool detection phase and uses the partial pointclouds and the RGB frames acquired during the scanning phase to locate tools correctly. The tool pose estimation process is made of 5 phases:

1. **Locate cluster's best partial pointcloud**

   The input clusters are likely to contain more error than the partial pointcloud whose geometric center is nearest to the cluster center. This is due to the fact that RGB-cameras tends to have better accuracy in the central regions than the peripheral

ones. This can be detected experimentally and also applies to stereo-based RGB-D cameras where in the central region, the views from both cameras are more aligned, leading to more accurate stereo matching. Also, distortion is less relevant in the central region of both cameras, making the triangulation estimation more accurate.

For this reasons, we will apply the cluster VOI on a partial pointcloud taken during scanning whose geometric center is the nearest to the cluster geometric center

2. **Locate cluster's best RGB frame**

   After having obtained the cluster's best pointcloud, we can proceed by taking the corresponding RGB frame. The RGB frame will be useful to accurately compute the position of tool's keypoints

3. **Keypoints position estimation**

   The estimation procedure is the result of the following assumptions:

   - **Keypoint detection is accurate on images**
     We will assume that tool keypoints present some color and geometric information that will be easier to locate with good accuracy on a RGB frame, rather than on a pointcloud. Pointclouds, in fact, tends to be far more sparse and less accurate (due to the voxel sizes and shape) than the pixel in image. For this reason, the keypoints detection will be performed on the RGB frames
   - **Keypoint's depth is accurate on pointclouds**
     Even though X,Y position is easier to locate with high accuracy on RGB frames, estimating the depth of such keypoints should be performed on point clouds. In fact, as previously stated, one hardware requirement for the depth sensor is the absence of a systematic over or under-estimation. That means, if we take a sufficiently high number of points describing a planar surface, the plane fitted to the averages of the points coordinates, will be very close to the real plane. This is a direct application of the Law of Large Numbers, which states that as the number of sampled points increases, the average of the sampled points converges to the true value

Now that we have clarified the assumptions, we can proceed with the algorithm:

Algorithm 6: 3D Point Cloud Processing Algorithm

Data: `image`, `pointcloud`, K
Result: Intersection point in 3D space

```
keypoint ← locateKeypoint(image)
neighbors ← getNeighbors(keypoint, image)
neighbors3D ← getNeighbors3D(neighbors, K, pointcloud)
plane ← fitPlaneWithLeastSquares(neighbors3D)
return interesection(plane, keypoint3D)
```

The target keypoint (it can more than one, but we will consider the case with one keypoint without loss of generalization), is located using a user-defined function. Then, we proceed by locating a set of nearby points, called neighbors, using a user-defined kernel (rectangular, circular or different). With the obtained points, we

recover their depth by projecting the pointcloud voxels in camera coordinates and remembering the depth

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Then, we proceed by fitting the plane to the obtained points. To fit this plane, we can use the Least Squares method. The algorithm is reported below:

Algorithm 7: Fit Plane to 3D Points Using Least Squares

Input: A set of 3D points $P = \{(x_i, y_i, z_i)\}_{i=1}^{n}$
Output: Coefficients of the fitted plane A, B, C, D such that
$Ax + By + Cz + D = 0$

Step 1: Convert points to a matrix form;

$$A = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{bmatrix}, \quad B = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix};$$

Step 2: Solve for the coefficients using least squares;

$$\begin{bmatrix} A \\ B \\ D \end{bmatrix} = \text{argmin}_{\hat{A},\hat{B},\hat{D}} \| A \begin{bmatrix} \hat{A} \\ \hat{B} \\ \hat{D} \end{bmatrix} - B \|^2;$$

Step 3: Normalize the coefficients to ensure $A^2 + B^2 + C^2 + D^2 = 1$;

$C \leftarrow -1$;

$\text{norm} \leftarrow \sqrt{A^2 + B^2 + C^2 + D^2}$;

$A \leftarrow \frac{A}{\text{norm}}, B \leftarrow \frac{B}{\text{norm}}, C \leftarrow \frac{C}{\text{norm}}, D \leftarrow \frac{D}{\text{norm}}$;

return *The plane coefficients* A, B, C, D

Then, we can proceed by computing the intersection between the line passing through the backprojected keypoint and the camera center and the previously computed plane. So, given the keypoints pixel coordinates (u, v), the 3D point corresponding to this pixel in camera coordinates is obtained by back-projecting the pixel using the inverse of K:

$$\mathbf{p}_C = K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

The equation of the line passing through the camera center $(0, 0, 0)$ and the point $\mathbf{p}_C$ is given by:

$$\mathbf{r}(t) = t \cdot K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \quad t \geq 0$$

In parametric form, this is:

$$\begin{bmatrix} X(t) \\ Y(t) \\ Z(t) \end{bmatrix} = t \cdot K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \quad t \geq 0$$

Now, we just need to compute the intersection point between the parametric line and the previously computed plane.

Given the parametric equation of the line:

$$\mathbf{r}(t) = \mathbf{P}_0 + t\mathbf{d}$$

where:

$$\mathbf{P}_0 = (x_0, y_0, z_0), \quad \mathbf{d} = (d_x, d_y, d_z)$$

And the equation of the plane:

$$Ax + By + Cz + D = 0$$

We can substitute the parametric line equation into the plane equation:

$$A(x_0 + td_x) + B(y_0 + td_y) + C(z_0 + td_z) + D = 0$$

Expand and solve for t:

$$t = \frac{-(Ax_0 + By_0 + Cz_0 + D)}{Ad_x + Bd_y + Cd_z}$$

Once t is known, we can substitute it back into the line equation to find the intersection point:

$$\mathbf{P}_{\text{int}} = (x_0 + td_x, y_0 + td_y, z_0 + td_z)$$

After that, we can retrieve the orientation of the intersection point computing the normal vector of the plane in $\mathbf{P}_{\text{int}}$.

If the denominator of the t-equation is zero, the line is either parallel to the plane (no intersection) or lies on the planes. Clearly, these two scenarios must be avoided. A more high-level description of the procedure is given in the figure 30

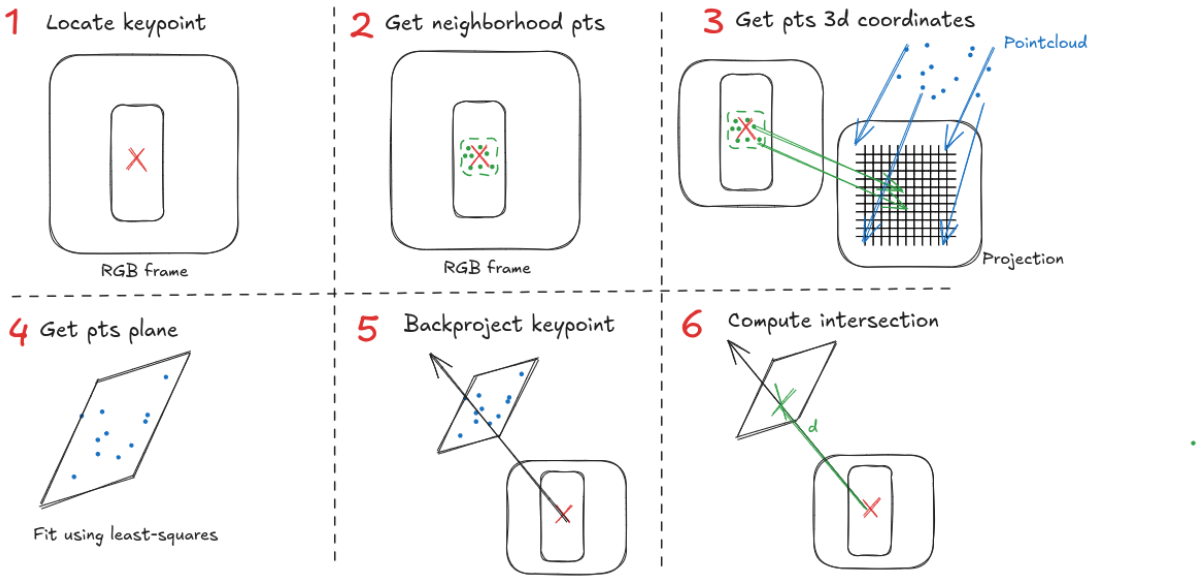


Figure 30: High level description of the pose estimation pipeline, having found the *best* pointcloud and corresponding RGB-frame

5.6.3 Services and Topics

- `/locator/locate`

  Triggers the `process` event, which will move the node state to `PROCESSING`

6 Experimental Analysis

To assess the accuracy of the previously described pipeline, an experiment has been conducted using industry-standard tools and easily accessible items. The source code of the pipeline is available on `https://github.com/AleNarder/master-thesis` and can be run locally via Docker container.

6.1 Experimental setup

The experiment has been conducted in the Kaigos SRL Operative Headquarters. The experimental cell has been built in order to simulate a press folding cell, where the press tools are placed linearly, with constrained pose, along a linear magazine placed in front of the robotic manipulator. In the next sub-sections, a brief description of the adopted tools will be given.

6.1.1 Robot

The robot used for the experiment is an Efort EFO-ER3-600 industrial robot. It is small a 6-dof robotic manipulator with a wrist payload of 3kg and a maximum reach of 593 mm. In figure 31

Figure 31: The EFO-ER3-600 industrial robot

From the official datasheet [15], this robot has a repeatability of ± 0.02 mm, that is, the euclidean distance between the requested cartesian pose and the real one can differ by 0.02 mm. The forward kinematics is performed directly by the on-board computer using calibrated Denavit-Hartenberg parameters.

6.1.2 RGB-D Camera

The RGB-D camera adopted for the experiment is a Realsense D-435i stereo-camera with built-in IMU controller. This camera is based on the very well-tested D430 stereo-vision

module which, from the official datasheet, reports a depth error of 2% and an ideal depth range from 0.3 to 3m.

Figure 32: Intel Realsense D435i stereocamera

The maximum RGB resolution offered by the RGB camera is 1280 × 720, with a FOV (Field of View) of 87° × 58°. This camera offers built-in ROS2 support as well as proprietary SDK (software Development Kit) with Python and C++ bindings.

6.1.3 Calibration object

In order to perform calibration tasks, the standard OpenCV chessboard has been used. The OpenCV chessboard is a nonspecular 9 x 6 chessboard with black and white squared cells. Each cell side has a dimension of 0.023m.

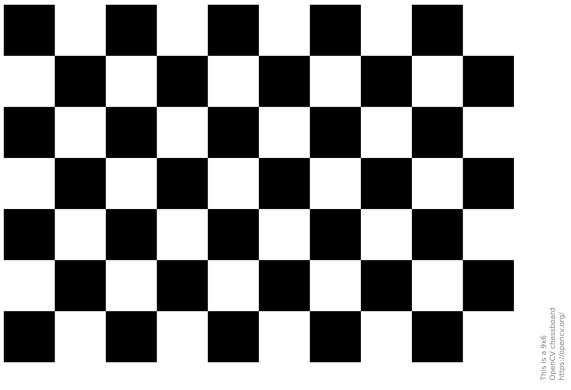


Figure 33: OpenCv calibration checkerboard

The pattern has been printed on a standard A4 paper sheet using an ink-jet printer. In this case, the measurement error of the printed copy has been assumed to be negligible.

6.1.4 Flange-Camera link

The Flange-Camera link has been made using a 3D printer, with standard PLA filament. The printer used to build the link is a Creality Ender 3, equipped with 0.4 mm nozzle. Because of the printer kinematics error and the nozzle size, the obtained tool is expected to have a measurement error along the shape and its depth of 0.5m. Referring to the

image 34, the two small holes located in the upper side of the image are used to attach the camera, the two holes with skews already mounted in the bottom section are used for the flange.

Figure 34: The 3D printed flange-camera (hand-eye) link

Other than the printing error, a small error is also introduced by the non-perfect alignment of the skews in the holes. This error, by the way, has been considered negligible and already included in the previously stated printing error.

6.1.5 Reference tools

The tools to detect and locate. Tools are made of steel and are used for metal bending.

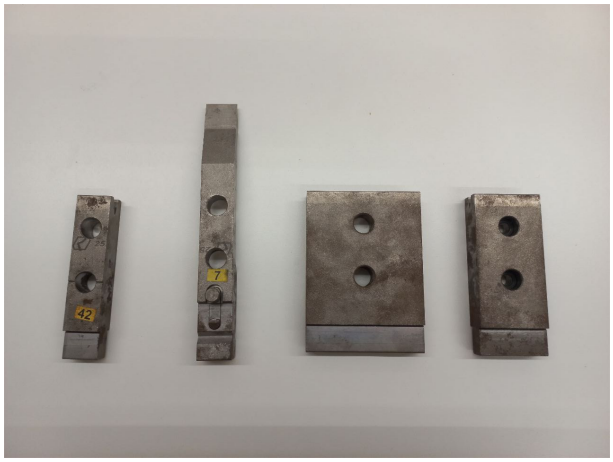


Figure 35: The reference tools

6.1.6 Experimental cell

Some snapshots of the entire cell are reported in figure 38

Figure 36: Cell seen from left

Figure 37: Cell seen from right

Figure 38: The experimental cell. The checkerboard position was moved between the photos due to singularities issues

6.2 Camera Calibration

The camera calibration has been performed using the previously mentioned calibration with the camera mounted on the flange of the robot. During the calibration process, the detected corners has been reproject on the target image to provide a visual feedback on the correctness of the calibration.

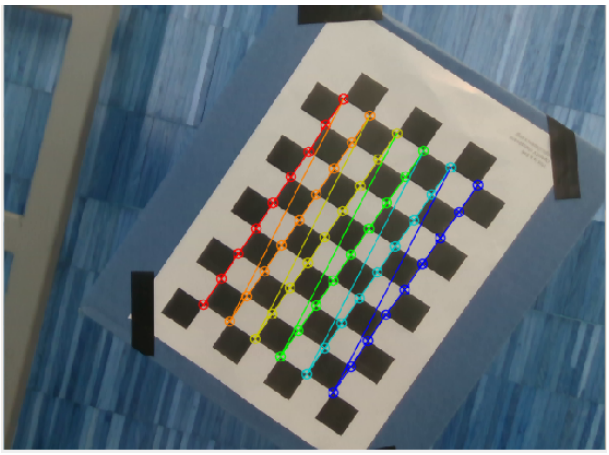


Figure 39: The checkerboard with detected corners

6.2.1 Settings

The calibration has been performed with a Laplacian variance threshold of 300, with K-Means clustering performed with $K = 30$ over a PCA-reduced dataset with 95% of explained variance

6.2.2 Final Results

The experimental results are reported in the table 2

Metric	Reprojection Error (px)
Mean Reprojection Error (MSE)	0.02
Max Reprojection Error	0.04
Min Reprojection Error	0.1

Table 2: Reprojection Error Metrics

The achieved reprojection error is below the pixel accuracy with a maximum reprojection error of 0.4px. These results are sufficiently accurate, so the the hand-eye calibration can be performed.

6.3 Hand-Eye Calibration

The hand-eye calibration has been performed using the reference calibration object and a sequence of robot poses. In this section, the adopted settings as well as the encountered issues will be presented

6.3.1 Settings and common issues

In a first attempt, hand-eye calibration was performed reaching a set of predefined poses following a continuous trajectory. This approach led to poor results, as shown in figure 40. The re-projected poses computed in the first iteration are spread over a large volume, with reprojection intervals along the axis spanning more than 10cm.

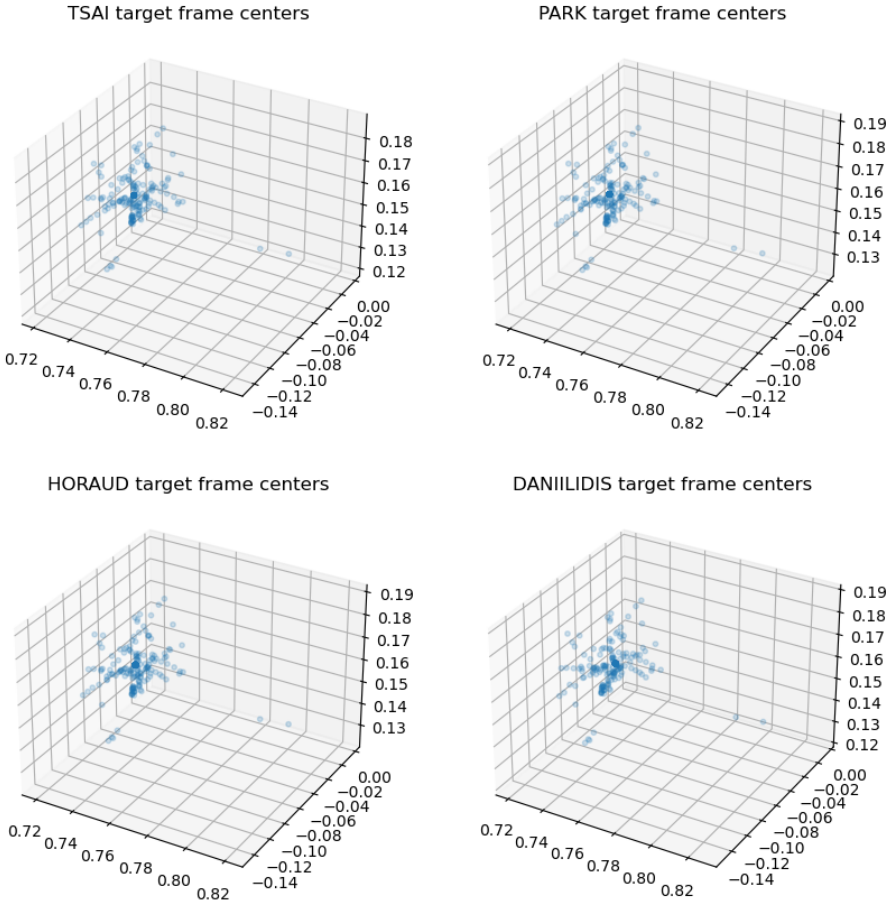


Figure 40: Reprojections without strict synchronization policy

Further investigations revealed that the timestamped queries used to perform the transforms lookup over the transforms buffer, using the RGB frame timestamp, were not sufficiently precise. In particular, the lookup query seemed to return delayed results and/or the RGB frame timestamp was not sufficiently accurate. To solve this problem, the maximum allowed flange speed threshold has been set to 0, i.e. the data gathering only works when the robot is not moving. The improvement obtained from this small update was significant, as shown in the first quadrant of figure 41.

In the first iteration, we can see how the volume spanned by the re-projected poses is dramatically smaller. The axes range now spans no more than 2cm, giving confidence

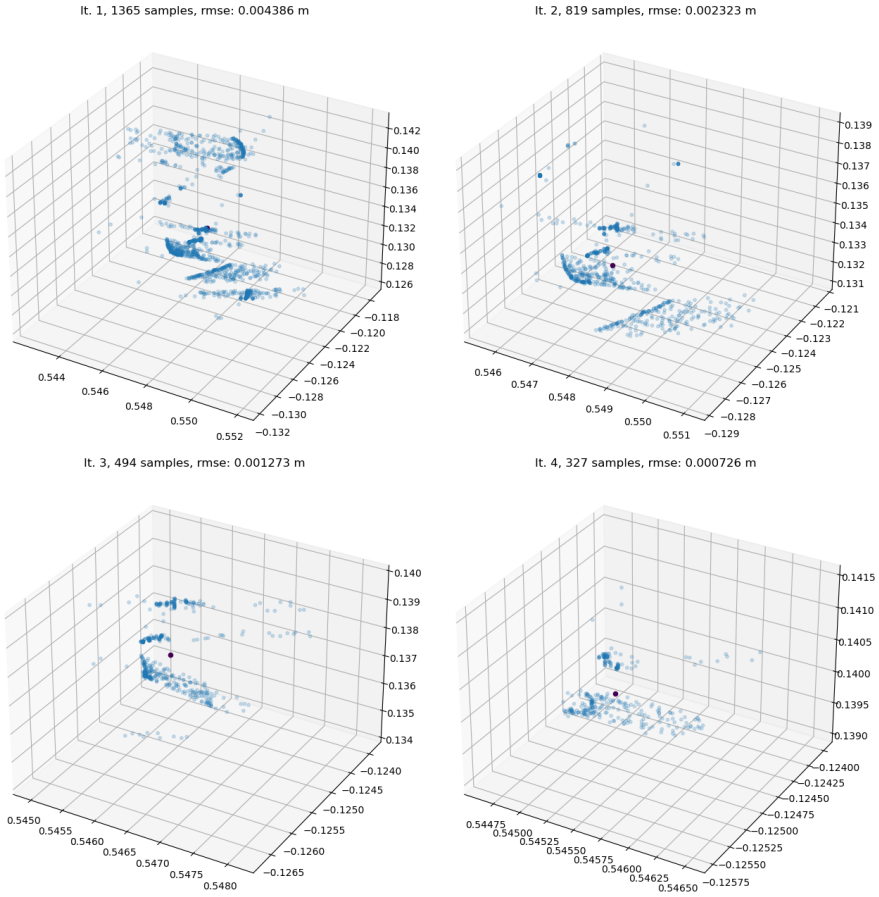


Figure 41: Reprojections with strict synchronization policy

about the quality of the data gathering. The hand-eye calibration was then performed using a minimum of 300 different poses for a maximum of 10 iterations. In figure 41, the re-projections obtained using the iterative hand-eye calibration method are reported, making evident how the more outliers are removed, the smaller the re-projection volume becomes.

6.3.2 Final Results

In table 3, the rooted mean squared distance of the re-projected poses to the cluster centroid ant the relative quaternion distance are reported for each iteration.

Iteration (it)	**Samples**	**RMSE**	Q_{dist}
1	1365	0.004386	0.01295
2	819	0.002323	0.00816
3	494	0.001273	0.00404
4	327	0.000726	0.00196

Table 3: RMSE values across iterations and samples

The re-projection errors decrease significantly over the iterations, leading to a very low final RMSE and small quaternion distance. The obtained values are sufficiently accurate for the next phase, i.e. the scanning phase.

6.4 Scanning

The scanning procedure has been performed following a linear scan in front of the set of tools. Tools have been placed along the same line, with constrained pose.

6.4.1 Settings and issues

Similarly to the hand-eye calibration, the scanning phase has been initially performed following a linear trajectory with a constant speed of 3cm/s. In this scenario, the tools were placed over the wall. The result of the scanning procedure is shown in **??**. It can be seen that even if the chromatic contrast has led to a good pointcloud resolution, the environment is not properly mapped.

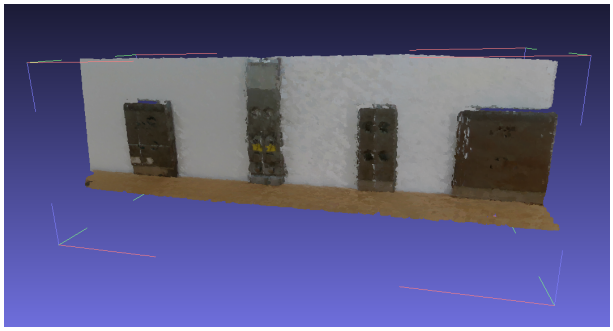


Figure 42: Pointcloud stitching without strict synchronization policy

Moreover, the position of the tools made the linear scanning difficult because of the reach of singularities along the trajectory. In order to solve this issues, the tools were moved in the robot base direction and the scanning max flange speed threshold was set to 0, i.e the scan proceeds reaching fixed stations, where it stops for 2 seconds, before continuing to the next station. The result of this updated policy is shown in figure 43. The environment now is correctly mapped, no singularities have been reached along the path, but the weaker color contrast has led to a more noisy pointcloud.

Figure 43: Pointcloud stitching with strict synchronization policy but no post-processing

6.4.2 Final Results

Even though the presence of noise in the starting pointcloud, the reduction of the partial pointcloud volume and the statistical outlier rejection followed by voxel downsampling has led to a good result, as shown in figure 44. Tools are correctly mapped, wrong interpolations have disappeared and flat surfaces are qualitatively smooth.

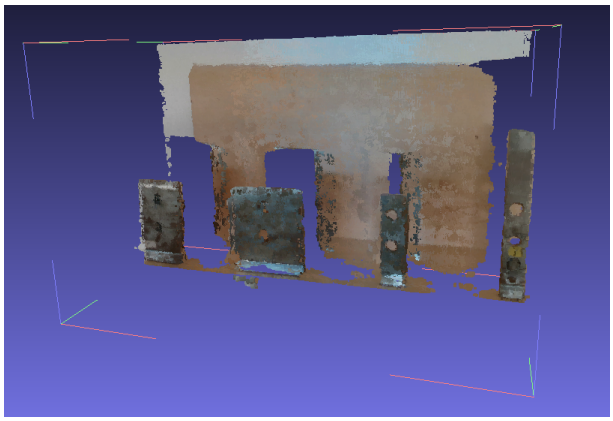


Figure 44: Post-processed pointcloud

6.5 Tool Detection

Tool detection has been performed using the stitched pointcloud generated in the previous step. Tool meshes have been loaded as .stl files. The .stl files have been created using Autodesk Fusion and the producer reference specifications

6.5.1 Settings and issues

Tool cluster detection has been performed using DBScan over a user-defined VOI (volume of interest), with `eps` set to 0.01 and `min_samples` set to 200. The VOI is shown in 45, whereas the clustering result is shown in 46.

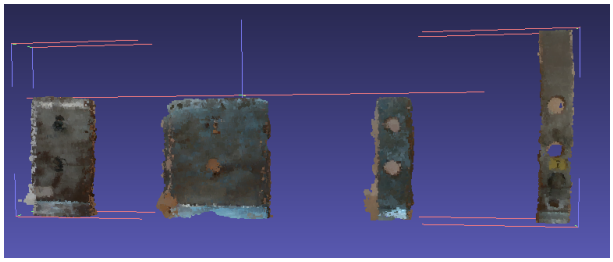


Figure 45: The VOI (Volume of Interest)

Clustering correctly detected the tool clusters, making the rasterization procedure straightforward, as shown in figure 47

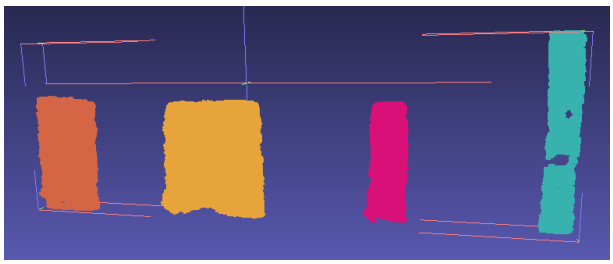


Figure 46: VOI after DBScan clustering

The matching via rasterization is presented in figure 47. Matching has been performed for each cluster and for each tool per cluster.

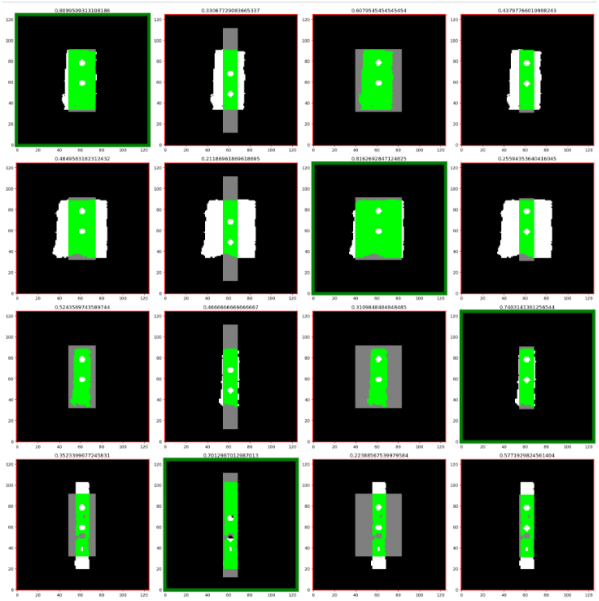


Figure 47: The clusters over meshes grid. For each entry, the grey shape represents the tool meshes, where the white is the cluster. Green area represents the overlapping area, which then is used to compute similarity

6.5.2 Final Results

The Jaccard similarities of each cluster to the tool meshes are given in table 4. A similarity can be considered a potential match only if the Jaccard similarity is above a given threshold (0.7 in this experiment)

Cluster	Tool 1	Tool 2	Tool 3	Tool 4
1	**0.8099**	0.3306	0.6079	0.4379
2	0.4849	0.2119	**0.8162**	0.2556
3	0.5243	0.4667	0.3109	**0.7403**
4	0.3523	**0.7013**	0.2239	0.5771

Table 4: Cluster tool data

All the clusters have been assigned to a given mesh label without ambiguities, even though the 4th cluster similarity is just below the threshold.

6.6 Tool Pose estimation

Tool pose estimation takes as input the clustered labels obtained in the tool detection step, as well as the partial pointclouds and RGb-frames taken during scanning

6.6.1 Settings and issues

The keypoint to detect in this experiment was the middle point of the segment starting from the upper circular hole to the lowest one. This keypoint is located on a flat surface, making it a good candidate for the pose estimation procedure. The keypoint has been located using `cv.HoughCircles` detector with ad-hoc params. In the image 48 the RGB-frames associated to partial pointclouds near to the cluster geometric center are shown, along with the detected keypoint

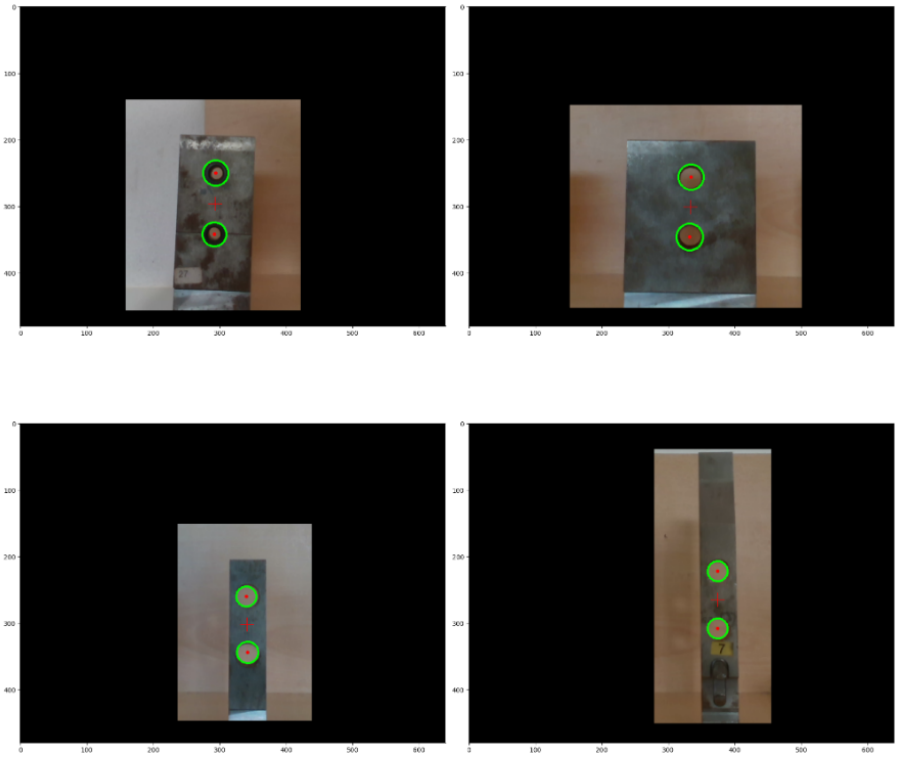


Figure 48: Keypoint detection

Then, we can proceed by computing the keypoint ray intersection with the cluster. We do so by computing the intersection with the ray and the plane fitted among a set of keypoint neighbors. For this case, a rectangular kernel centered on the keypoint pixel with a size of 11 x 11 pixel has been used.

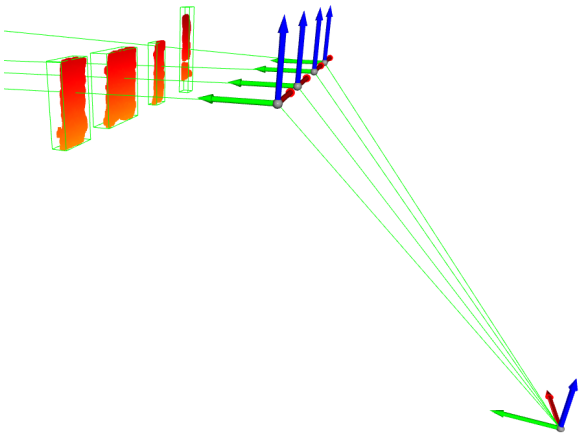


Figure 49: The lowest frame helper represents the robot base frame. The green lines ending on the camera frames set represent the camera position with respect to the base frame. The green rays from the camera frame crossing the clusters are the back-projected rays intersecting the keypoint pixel coordinates

6.6.2 Results

In table 5 the detected poses are reported whereas int 6 we have the ground truth poses. The ground truth poses have been taken using the robot in manual control and a caliber. For both tables roll, pitch and yaw have been reported in degrees while X, Y, and Z in meters, with respect to the robot base frame

Tool	$\hat{X}$	$\hat{Y}$	$\hat{Z}$	$\hat{W}$	$\hat{P}$	$\hat{R}$
Tool 1	-0.3038	0.7174	0.5387	-179.12	1.90	1.13
Tool 2	-0.1527	0.7162	0.5370	-178.82	0.87	0.67
Tool 3	0.0107	0.7168	0.5371	-177.51	1.06	1.50
Tool 4	0.1526	0.7176	0.5677	-179.00	1.41	1.92

Table 5: Estimated tools positions and orientations

Tool	X	Y	Z	W	P	R
Tool 1	-0.3020	0.7170	0.5380	-179.12	0.16	-0.09
Tool 2	-0.1500	0.7170	0.5380	-179.24	0.16	-0.09
Tool 3	0.0100	0.7170	0.5380	-179.16	0.16	-0.09
Tool 4	0.1500	0.7175	0.5680	-179.28	0.16	-0.09

Table 6: Ground truth tools positions and orientations.

7 Conclusions and further improvements

In this work, an end-to-end tool pose estimation pipeline for robotic manipulators with RGB-D camera mounted on the flange has been presented. The proposed pipeline offers a simple yet effective framework to locate tools with constrained poses in the surrounding environment using both volumetric and color information. The experimental results have validated the good design of the system in different aspects: the offline and step-by-step nature of the pipeline have greatly simplified the tuning and validation process, making easy to spot inconsistencies coming from the data sources. The iterative hand-eye calibration had proven to be effective even in the presence of outliers, but its ability to produce consistent results heavily depends on the quality of the gathered data, on both resolution and synchronization. In particular, synchronization can also lead to inconsistent results during stitching, which is a fundamental step for tool detection and localization. On the other hand, noise and wrong interpolations accumulated during stitching have been effectively removed using a two-stage noise-removal routine and exploiting the geometric properties of the stereo camera. Tool detection has shown to be particularly effective, with a detection rate of 100% and good generalization. However, the rasterization-based matching could lead to ambiguities if non-planar surfaces are present, requiring more attention to the tool-posing strategy. In the end, experimental results have validated the object pose estimation procedure, resulting in low error on both position (less than 1mm on each axis on average) and orientation of the tools (less than 2 degrees on average).

Even though an end-to-end solution has been proposed, the capabilities of the system could be further improved with some additional features. In particular, the constrained pose requirement could be relaxed using 2D or 3D-matching algorithms, assuming good environmental mapping and resolution. Moreover, the tool localization procedure could be improved by introducing a polynomial surface fitting and an iterative depth estimation procedure, which applies the keypoint localization and depth estimation through a set of progressively near poses.

Summing up, the proposed pipeline has shown to be a robust and effective solution for both solving the robot hand-eye calibration and the object pose estimation, offering a plug-play system capable of solving different sub-problems. The step-by-step nature of the pipeline allows great customization and flexibility, other than powerful representations, making the system a good foundation model also for use-case specific scenarios.

References

[1] Richard Paul. *Robot manipulators: mathematics, programming, and control: the computer control of robot manipulators*. Cambridge, Massachusetts: MIT Press, 1981. ISBN: 978-0-262-16082-7.

[2] J. M. McCarthy. *Introduction to Theoretical Kinematics*. Cambridge, Massachusetts: MIT Press, 1990.

[3] Seoul National University. *Manipulator Kinematics*. Accessed: 22-Sep-2024. 2011. URL: https://ocw.snu.ac.kr/sites/default/files/NOTE/Chap03_Manipulator%20kinematics.pdf.

[4] Universal Robots. *DH Parameters for calculations of kinematics and dynamics*. Accessed: September 15, 2024. URL: https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/.

[5] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. 2nd. New York, NY, USA: Cambridge University Press, 2004.

[6] Clément Ernould et al. "Chapter One - Measuring elastic strains and orientation gradients by scanning electron microscopy: Conventional and emerging methods". In: ed. by Martin Hÿtch and Peter W. Hawkes. Vol. 223. Advances in Imaging and Electron Physics. Elsevier, 2022, pp. 1–47. DOI: https://doi.org/10.1016/bs.aiep.2022.07.001. URL: https://www.sciencedirect.com/science/article/pii/S1076567022000544.

[7] OpenCV Contributors. *Camera Calibration and 3D Reconstruction*. Accessed: 2024-09-24. 2024. URL: https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga3207604e4b1a1758aa66acb6ed5aa65d.

[8] Qi Zhang et al. "Rolling guidance filter". In: *Computer Vision–ECCV 2014*. Springer, 2014, pp. 815–830.

[9] Jean-Yves Bouguet. *Camera Calibration Tool Box for MATLAB [EB/OL]*. Accessed: 2024-09-24. 2004. URL: http://www.vision.caltech.edu/bouguetj/calib_doc/.

[10] R. Y. Tsai and R. K. Lenz. "A new technique for fully autonomous and efficient 3d robotics hand/eye calibration". In: *IEEE Transactions on Robotics and Automation* 5.3 (June 1989), pp. 345–358.

[11] Radu Horaud and Fadi Dornaika. "Hand-eye calibration". In: *Int. J. Rob. Res.* 14.3 (June 1995), pp. 195–210.

[12] F. C. Park and B. J. Martin. "Robot sensor calibration: solving ax=xb on the euclidean group". In: *IEEE Transactions on Robotics and Automation* 10.5 (Oct. 1994), pp. 717–721.

[13] Konstantinos Daniilidis. "Hand-eye calibration using dual quaternions". In: *International Journal of Robotics Research* 18 (1998), pp. 286–298.

[14] Filippo Bergamasco. *Epipolar Geometry - 3DCV course*. Accessed: 2024-09-26. 2019. URL: https://www.dsi.unive.it/~bergamasco/teachingfiles/cvslides2019/15_Epipolar_geometry.pdf.

[15] SensorLine. *ER3 600*. Accessed: 2024-09-29. 2023. URL: https://sensorline.biz/wp-content/uploads/2023/11/er3-600.pdf.

[16] John J. Craig. *Introduction to Robotics: Mechanics and Control*. 3rd. Prentice-Hall, 2004.

[17] J. M. McCarthy and G. S. Soh. *Geometric Design of Linkages*. 2nd. Springer, 2010.

[18] Nicolas Andreff, Radu Horaud, and Bernard Espiau. "On-line hand-eye calibration". In: *Proceedings of the 2Nd International Conference on 3-D Digital Imaging and Modeling, 3DIM'99*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 430–436.

[19] Bruno Siciliano. *Convenzione di Denavit-Hartenberg*. Accessed: September 15, 2024. URL: https://moodle.calvino.ge.it/pluginfile.php/4337/mod_resource/content/0/DH%20essentials.pdf.

[20] Paul van Walree. *Distortion. Photographic Optics*. Archived from the original on 29 January 2009, Retrieved 2 February 2009. 2009. URL: http://www.vanwalree.com/optics/distortion.html.

[21] Wikipedia contributors. *Robot kinematics — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-September-2024]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Robot_kinematics&oldid=1152657384.

[22] Hedi Vision. *Pinhole Camera Model*. Accessed: 16-September-2024. 2019. URL: https://hedivision.github.io/Pinhole.html.

List of Figures

List of Tables