POLITECNICO DI TORINO

Master Degree in Mechatronic Engineering

Master Degree Thesis

Artificial Intelligence and Computer Vision Techniques for Human-Robot Interaction

Three different applications to *Universal Robots* collaborative robots

Advisor
Prof. Marina Indri

Candidate
Alice Maritato

Corporate supervisor
Universal Robots
Eng. Paolo Bassetti

October 2020

Abstract

The aim of this thesis is to investigate techniques to improve the interaction between humans and collaborative robots (i.e. robots which share the workplace with humans without barrier and without exposing the human to risks) exploiting *Artificial Intelligence* and *Computer Vision techniques*.

To highlight the effectiveness and the potential of Computer Vision (CV) applied to *Robotics* and to analyze how the *Human-Robot Interaction* might be powered leveraging the CV expertise, three cases of study, set in three different contexts, are developed for the UR5 cobot (developed by *Universal Robots*) and the required algorithms are developed using state-of-the-art methods. The first is set in a television studio where the television presenter moves in the studio and the robotic arm follows him using a face-tracking algorithm. The second is set in an industrial context where an operator has to process workpieces which are numbered from 1 to 5. The operator is able to select the workpiece to be processed using gestures, and the camera-equipped robot responds accordingly. The third is set in a hospital: a patient is in its bed and, unable to move, he can communicate with the robot with gestures and ask for drugs, water or telephone.

These cases of study are simulated in the *Polyscope* simulation environment, the proprietary software of Universal Robots, and *RoboDK* which is a universal environment for robot simulation. The computer code required for CV and AI tasks, which is developed in *Python* using the *OpenCV* library, is thoroughly analyzed in this work, as well as the state-of-the art models that were exploited and the related theory.

Acknowledgements

First of all, I would like to express my sincere gratitude to the people who provided me guidance during the months in which I worked on my Master Thesis. I would like to thank my supervisor Marina Indri for her willingness and for the thoughtful feedback and comments. I am also thankful to *Universal Robots* and, in particular, Paolo Bassetti and Andrea Macheda, for the challenging meetings and conversations, for giving me new ideas, for offering different viewpoints and for the passion which they inspired me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction and state of the art

Human-Robot Interaction is a wide and multidisciplinary field which stems from the need for better integration between robots and humans. Nowadays, due to their intrinsic advantages over humans, robots are ubiquitous in an increasingly large number of fields, such as manufacturing, medical industry, search and rescue operations, space exploration, and more.

The many advantages of robots include their preciseness, efficiency, high payload and expendability in hazardous situations. However, robots also have their drawbacks: traditionally, they have been unable to respond to unexpected situations and for this reason they are also potentially dangerous for humans that may invade their workspace without all due precautions. This has limited their seamless integration in manufacturing processes as well as their usage in more domestic environments, where they would have to interact with common people rather than trained operators.

In fact, robots have been traditionally designed to work in their own workspace in the absence of humans, and safety sensors and physical barriers are then used to ensure that humans do not interfere with their operation. In these situations, communication is therefore limited to dedicated interfaces to be typically used by skilled operators.

However, robots have become sophisticated enough to theoretically enable their usage in direct collaboration with humans: this particular kind of robots is often refereed to as collaborative robots or *cobots*. Differently from regular robots, which are confined to specific tasks to be executed in a dedicated environment, cobots are able to effectively and safely work together with humans in a shared environment.

The applications of cobots are manyfold: for example, they have opened the door to unexplored applications in medicine, such as teleoperations with surgical robots, in dangerous search and rescue operations. Coming to simpler scenarios, cobots might help patients in hospitals improving their life and making them more independent, or again it can help to film a news report in dangerous situations, such as an epidemic.

The industrial partner of this thesis work, *Universal Robots*, is among the companies that is actively developing collaborative robots. In particular, instead of focusing on large industries only, it works with small and medium industries as well as non-industrial (such as science and research) entities to develop solutions which can help them through the integration of cobots in human-friendly work environments.

The interaction between robots and humans might happen in different ways, according with the location (sharing or not the same workplace) and the timing (simultaneous or successive processes) in which robots and humans interact.

In recent years, the increasing diffusion of cobots has fueled interest for HRI, capturing the interest of researchers of many different fields. Among these fields, *Artificial Intelligence* (AI) and *Computer Vision* (CV) can give an important contribution to improve and to explore the possible applications of Human-Robot Interaction making it smarter. In an industrial context,

the flow of processes may increase in efficiency and may be optimized if an operator wants to communicate with robots through gestures the action to perform instead of button-pushing to run a certain program on the teach pendent of the robot. Nowadays, face detection and gestures recognition are two research fields which are deeply studied. Currently AI algorithms allow to detect faces in images or video regardless skin color, lighting conditions and head pose. The face detection is the first step to perform more complex operations such as face tracking or face recognition. Moreover, gesture recognition allows the user to interact with a certain devices or robots without touching any part of them.

The application of face tracking and gesture recognition algorithms to improve the interaction between humans and collaborative robots is an interesting and active research topic as it enables new ways to communicate with robots which were typically precluded to them. For example, the authors in [9] developed a solution which combines face tracking and gesture recognition techniques. Face detection is constantly performed, while gesture recognition is only performed if both a face and a hand which can be associated to that face are detected. Thus, the robot is not only able to receive inputs from an operator trough gestures, but it is also able to discriminate between the operator and simple by-passers. Several image segmentation phases are used to discriminate skin regions from the other objects in the frame. Then, the image is encoded using Haar-like features [32], which are then used to perform recognition with a machine learning algorithm called *Adaboost* [15].

An example of traditional image processing techniques for gesture recognition can be found in [34]. The authors designed a gesture recognition algorithm which controls the robot navigation using four dynamic gestures ('Go Right', 'Go Left', 'Go Forward', 'Go Backwards') and two static gesture ('Stop' and 'Turn around'). To test the effectiveness of this gesture recognition algorithm, movement which the robot has to perform is Graphical User Interface on a computer.

Another interesting application is presented in [6]. The authors used face recognition and a hand detector to develop a robotic puppet which is able to play games which are designed to improve social interaction skills in autistic children.

In [42] the authors investigated how to control the motion of a camera using face tracking. The system they developed is able to follow a face maintaining it in the center of the camera frame. They used the Viola-Jones algorithm, which is one of the most popular face detection algorithms [28].

In [20], the authors studied the interaction between humans and the robot Aibo, a robotic dog developed by Sony. To send data to the robot the TCP/IP wireless network was used. To recognize the gestures, the BGR frame is converted in YIQ and then the classification is performed by the *subspace method* which allows perform a fast classification of images extracting only the most numerous property of each class. Gestures such as 'One', 'Two', 'Three', 'Two Hand' corresponds to actions to perform by the robot such as stand up, walk forward or backward and to sit the robotic dog.

The aim of this work is to explore some relevant concepts in which mixing HRI and CV might generate interesting outcomes helping a television presenter, the operator of an industry or the patient of a hospital. The following three cases of study are analyzed:

- **Face-tracking in a television studio:** the presenter moves around in the television studio and the robot arm, provided by camera, follows his movement and focuses on him. In particular, the movement of the robot is simulated in *Polyscope*, while a personal computer (PC) runs a code written in *Python* which acquires the video stream from the PC webcam, detects the face through a face-detection algorithm, computes the displacement which the robotic arm has to perform and the pose that the end-effector of the robot has to reach. Then, it sends data to the simulated robot using the *Real-Time Data Exchange* interface, an interface developed by Universal Robots which allows real-time

communication between robots and external interfaces.

- **Gesture recognition in an industrial context:** the workplace consists in an input station, a workbench in which the operator can process the workpieces and an output station. At first, all workpieces are in the input station. Workpieces are numbered. The operator performs a gesture, which is the mimic of a number and which corresponds to the piece which he has to process, in front of the camera. The robot recognizes the gesture, grabs the correspondent piece and brings it to the operator. When the operator has finished to process the piece, the robot carries the workpiece to the output station. The movement of the robot is simulated on both Polyscope and *RoboDK*, which provides a richer simulation environment, also enabling the simulation of the grabbing and dropping operations of the end-effector. The webcam of the PC is used to capture the video stream used for the gesture recognition which is performed on the same PC through a Python script. The code performs hand detection followed by the application of several Computer Vision Techniques which allow to obtain the Convex Hull, the Convexity Defects and the Center of the Contours characterizing the shape of the hand. Based on these elements, gesture recognition is performed.

- **Gesture recognition in a hospital:** A patient, unable to move, is in the bed. Around him, there are a medicine cabinet on which a package of medicine is located and a bedside table on which there are a mobile phone and a glass of water. The patient performs a gesture in front of a camera and the robot grabs the object which corresponds to the gesture, brings it to the patient and places it on the tray on the bed. The gesture recognition and the simulation environment used are the same as for the second case of study.

The contents of this thesis are structured as follows. Chapter 2 introduces some basic concepts about robot kinematics, a technical description of the *UR5* robotic arm, and communication interfaces that are relevant for this thesis. Chapter 3 illustrates some tools that were used in this work, such as the *Polyscope* and *RoboDK* simulation environments and the *OpenCV* computer vision library. Chapter 4 presents theoretical aspects of the computer vision and artificial intelligence techniques that were used to develop the applications of Chapter 5. Finally, Chapter 5 details the programs developed for each of the cases of study developed for this thesis.

Chapter 2

Introduction to Robotics and to the Universal Robots world

In this chapter, some preliminary information about robots is presented. First, some basic nomenclature for robots is given and the mathematics that describes robot position and movements is introduced. Then, other concepts associated to Robotics, such as *cobots* and *human-robot interaction* are explained. Finally, a brief overview about *Universal Robots* and an analysis of the UR5 robot arm are presented.

2.1 Robotics preliminary concepts

In this section the key aspects of robot mechanical structure and robot kinematics are analyzed in order to provide the reader all the concepts which will be referred to in this work.

2.1.1 Robot Mechanical Structure

Robots can be classified according to the features of their base:

- **Robot manipulators:** they have a fixed base, as Figure 2.1 shows;
- **Mobile robots:** they have a mobile base, as Figure 2.2 shows.

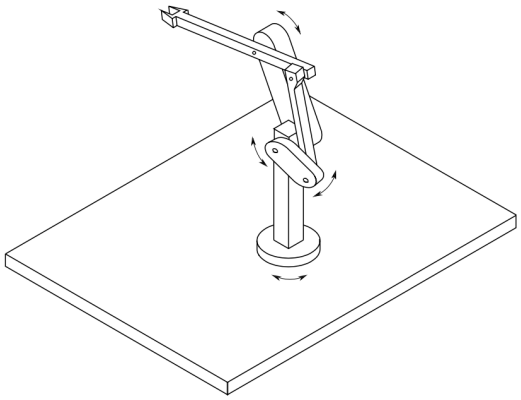

Figure 2.1: Robot manipulator. Source:[47].

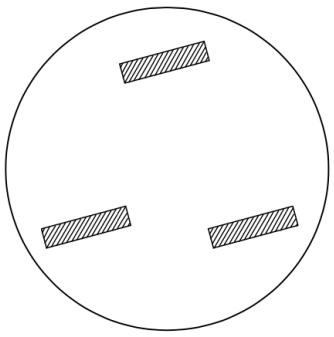

Figure 2.2: Mobile robot. Source:[47].

Robot Manipulators

The mechanical structure of a robot manipulator is composed by *links*, which are rigid bodies interconnected by articulations called *joints*. The *end effector* is a device which allows the robot to interact with its environment and to perform a given task. It consists of a gripper or tools for cutting, drilling, welding, etc. [47]. Robot manipulators can have two different types of structure:

- **Open kinematic chain:** the kinematic chain is open if there is only one sequence which connects the robot links, as Figure 2.3 shows;
- **Closed kinematic chains:** the kinematic chain is closed if the link sequence forms a loop, as Figure 2.4 shows.

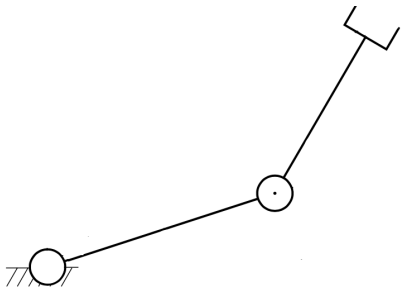

Figure 2.3: Two-link open-chain planar arm. Source:[47].

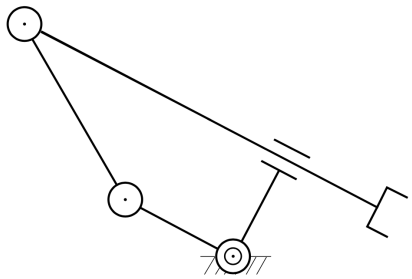

Figure 2.4: Four-link closed-chain planar arm. Source:[47].

The number of degrees of freedom, or DOFs, should be sufficient and properly distributed along the mechanical structure, in such a way that the robot is able to perform a given task. If the number of degrees of freedom is higher than the task variables needed to accomplish a task, the manipulator is defined as kinematically redundant.

The space in the environment which is reachable by the end-effector is called *workspace* and it is influenced by the manipulator structure as well as the mechanical joint limits. An example of workspace is shown in Figure 2.5.

2.1.2 Robots Kinematics

In this section, some basic aspects of the robot kinematics are recalled, such as what is a *pose*, the difference between *tool* and *joint space* and *inverse kinematics*. All these topics are preparatory to understanding the next chapters.

Pose

Knowing the position and the orientation of the end-effector is fundamental to manipulate an object in space. Given a reference frame (RF) the position of a rigid body in the 3D space with respect to this reference frame can be described by position and orientation [47]. The formal definition of pose is [7]:

$$\boldsymbol{p}(t) \stackrel{\text{def}}{=\!=} \begin{bmatrix} p_1(t) \\ p_2(t) \\ p_3(t) \\ p_4(t) \\ p_5(t) \\ p_6(t) \end{bmatrix} = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ \alpha_1(t) \\ \alpha_2(t) \\ \alpha_3(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}(t) \\ \boldsymbol{\alpha}(t) \end{bmatrix} \tag{2.1}$$

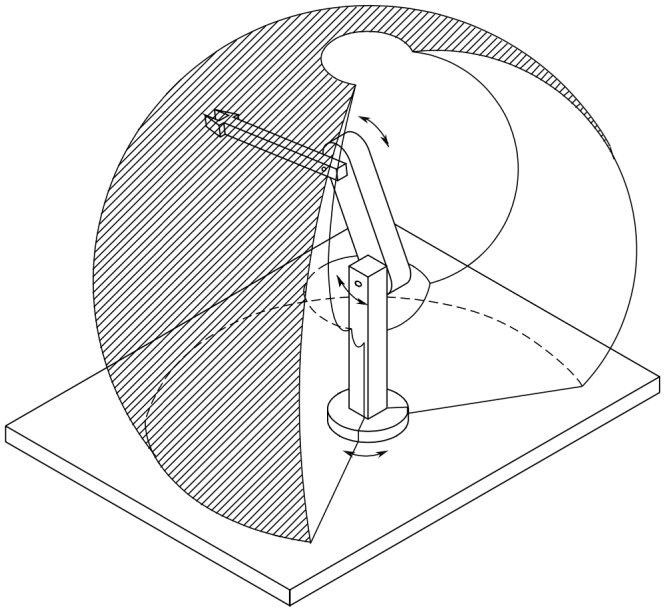

Figure 2.5: Workspace of an anthropomorphic manipulator. Source:[47].

The $\boldsymbol{x}$ vector defines the position, while the $\boldsymbol{\alpha}$ vector describes the orientation.

Translations

The term *rigid motion* indicates the displacement of a rigid body. This displacement of a rigid body in 3D space is the composition of a rigid translation along a line and a rotation along an axis parallel to this line.

Given a vector $\boldsymbol{v} = [v_1\ v_2\ v_3]^T$, the rigid translation along a specified direction given by the vector $\boldsymbol{t} = [t_1\ t_2\ t_3]^T$ is described by the operator $Trasl(\boldsymbol{v}, \boldsymbol{t})$ [7]:

$$Trasl(\boldsymbol{v}, \boldsymbol{t}) \equiv \boldsymbol{v}^t \stackrel{\text{def}}{=\!=} \begin{bmatrix} v_1 + t_1 \\ v_2 + t_2 \\ v_3 + t_3 \end{bmatrix} = \boldsymbol{v} + \boldsymbol{t} \tag{2.2}$$

Rotations

Given a square matrix $\boldsymbol{R}$ and a generic vector $\boldsymbol{v} = [v_1\ v_2\ v_3]^T$, the rotation is described by the operator $Rot(\boldsymbol{v}, \boldsymbol{R})$ [7]:

$$Rot(\boldsymbol{v}, \boldsymbol{R}) = \boldsymbol{R}\boldsymbol{v} \tag{2.3}$$

To make more than one rotation, each of which is represented by a matrix $Rot(\boldsymbol{u_i}, \boldsymbol{\theta_i}) = \boldsymbol{R}_i^{i-1}$ with respect the the reference frame obtained from the previous rotation, the total rotation is given by:

$$Rot(\boldsymbol{u}_1, \boldsymbol{\theta}_1) Rot(\boldsymbol{u}_2, \boldsymbol{\theta}_2) \cdots Rot(\boldsymbol{u}_N, \boldsymbol{\theta}_N) = Rot(\boldsymbol{u}, \boldsymbol{\theta}) \tag{2.4}$$

where $\boldsymbol{\theta}$ is not the sum of the single angles and $\boldsymbol{u}$ is the unit vector. The matrix which represents the global rotation will be:

$$Rot(\boldsymbol{u}, \boldsymbol{\theta}) \leftrightarrow \boldsymbol{R}_N^0 \stackrel{\text{def}}{=\!=} \boldsymbol{R}_1^0 \boldsymbol{R}_2^1 \cdots \boldsymbol{R}_N^{N-1} \tag{2.5}$$

Operational and joint space

The position and the orientation of the end-effector can be described by the vector $\mathbf{x_e}$ which is defined in the operational space (or *tool space*), which is the space where the manipulator task is specified.

$$\boldsymbol{x}_e = \begin{bmatrix} \boldsymbol{p}_e \\ \boldsymbol{\psi}_e \end{bmatrix} \tag{2.6}$$

where $\boldsymbol{p}_e$ is the end-effector position and $\boldsymbol{\psi}_e$ its orientation.

The *joint space* (or *configuration space*) describes the space in which the vector $\boldsymbol{q}$ of the joint variable is defined [47]:

$$\boldsymbol{q} = \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix} \tag{2.7}$$

Homogeneous coordinates

Homogeneous coordinates, or *perspective coordinates*, allow to use a unique operator for translations and rotations. Given a point P in 3D space, the homogeneous translation vector $\tilde{\boldsymbol{v}}$ is defined as [7]:

$$\tilde{\boldsymbol{v}} \stackrel{\text{def}}{=\!=} \begin{bmatrix} \lambda p_1 \\ \lambda p_2 \\ \lambda p_3 \\ \lambda \end{bmatrix} = \lambda \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} \tag{2.8}$$

where λ is a scale factor. Generally, λ is set to 1 according to the representation adopted to study the roto-translations of a rigid body in a 3D space, thus obtaining:

$$\tilde{\boldsymbol{v}} \stackrel{\text{def}}{=\!=} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} \tag{2.9}$$

A roto-translation can be represented by a homogeneous matrix:

$$\boldsymbol{T}_m^0 \stackrel{\text{def}}{=\!=} \begin{bmatrix} \boldsymbol{R}_m^0 & \boldsymbol{t}_m^0 \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tag{2.10}$$

where $\boldsymbol{R}_m^0$ represents the rotation, $\boldsymbol{t}_m^0$ represents the translation and $\boldsymbol{0}^T \stackrel{\text{def}}{=\!=} [0\ 0\ 0]$.

The homogeneous transformation matrix of a pure rotation transformation is:

$$\boldsymbol{T}(\boldsymbol{R}) \equiv \boldsymbol{T}_{\boldsymbol{R}} \stackrel{\text{def}}{=\!=} \begin{bmatrix} \boldsymbol{R} & \boldsymbol{0} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tag{2.11}$$

The homogeneous transformation matrix of a pure translation transformation is:

$$\boldsymbol{T}(\boldsymbol{t}) \equiv \boldsymbol{T}_{\boldsymbol{t}} \stackrel{\text{def}}{=\!=} \begin{bmatrix} \boldsymbol{I} & \boldsymbol{t} \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tag{2.12}$$

where $\boldsymbol{I}$ is the identity matrix.

To correctly perform a roto-translation transformation which is the composition of translations and rotations, some rules have to be followed: calling $\boldsymbol{T}(i)$ the matrix obtained after the i displacement, at first i has to be set to zero ($i = 0$) and $\boldsymbol{T}(0) = \boldsymbol{T}$. According to the type of reference frame:

- If the roto-translation $\boldsymbol{T}^i$ for $i = 1, ..., n$ is defined with respect to the fixed reference frame: $\boldsymbol{T}(i) = \boldsymbol{T}^i\ \boldsymbol{T}(i-1)$
- If the roto-translation $\boldsymbol{T}^i$ for $i = 1, ..., n$ is defined with respect to the mobile reference frame: $\boldsymbol{T}(i) = \boldsymbol{T}(i-1)\ \boldsymbol{T}^i$

2.2 Cobots

In 1997 the term *cobot* appeared for the first time in a United States patent. Two professors of the Northwestern University, James E. Colgate and Michael A. Peshkin, described its features in this way: *"In place of the actuators that move conventional robots, however, cobots use variable transmission elements whose transmission ratio is adjustable under computer control via small servomotors. Cobots thus need few if any powerful, and potentially dangerous, actuators. Instead, cobots guide, redirect, or steer motions that originate with the person. A method is also disclosed for using the cobot's ability to redirect and steer motion in order to provide physical guidance for the person, and for any payload being moved by the person and the cobot. Virtual surfaces, virtual potential fields, and other guidance schemes may be defined in software and brought into physical effect by the cobot"* [10].

Nowaday, a collaborative robot, or cobot, is a a robot which can operate in collaboration with a human operator. Since cobot and humans share the workspace or they are in close proximity, to constrain and guide the motion virtual surfaces are set up to guarantee the safety of humans [39].

In 2004, the first cobot on the market, LBR 3, was realized by KUKA. In 2008, Universal Robots, realized its firts cobot, UR5. After that, in 2012 Rethink Robotics launched to market its cobot Baxter and in 2015 both FANUC and ABB realized its first cobot.

Given the increasing diffusion of cobots on the market, in 2016 the International Organization for Standardization (ISO) made regulations about them. In particular, *"ISO/TS 15066:2016 specifies safety requirements for collaborative industrial robot systems and the work environment and supplements the requirements and guidance on collaborative industrial robot operation given in ISO 10218-1 and ISO 10218-2; it applies to industrial robot systems as described in ISO 10218-1 and ISO 10218-2. It does not apply to non-industrial robots, although the safety principles presented can be useful to other areas of robotics"* [14].

2.3 Human Robot Interaction

Since the 1940s, when Isaac Asimov coined the term *robotics*, a recurrent question arose about the relationship between humans and robots [5]. Human Robot Interaction, or HRI, studies how design and evaluate robotic systems which are used by or with humans. The interaction between humans and robots requires that a communication happens. This communication is influenced by proximity between robot and human [19]. Considering the proximity, three categories of interaction can be defined:

- **Remote HRI:** human and robot are physically in remote places. This type of operation, called tele-operation, could involve dangerous or scenarios which are not accessible to humans;

- **Co-located HRI:** human and robot are in a shared space and they interact without a physical contact;
- **Physical HRI:** human and robot are in a shared space and they interact with a physical contact [30].

The types of human robot interaction most commonly used is the co-located HRI. In fact, it can be used in a wide range of operations. For example, robots can perceive the presence of humans, map the movement of a human, recognize speech through sensors. In this case, based on whether the workspace is shared or not and on the timing, i.e the presence at the same time of the human which works and a robot which moves [48]. As Table 2.1 shows, four types of operation might be distinguished:

- **No interaction:** the operator and the robot do not work in the same workspace in the same time;
- **Coexistance:** the operator and the robot work simultaneously, but they do not share the workspace;
- **Cooperation:** the operator and the robot work sharing the same workspace, but not at the same time;
- **Collaboration:** the operator and the robot work sharing the same workspace at the same time.

Application	**Different workspace**	**Shared workspace**
Sequential processing	No interaction	Cooperation
Simultaneous processing	Coexistence	Collaboration

Table 2.1: Different types of HRI based on the space and time parameters. Source:[48]

The HRI is used in many fields of application: in an industrial context, such as a manufacturing plant, collaboration between humans and robots can use the capabilities of both to optimize the production chain, for example exploiting the preciseness and the ability to move heavy objects of robots [23]. However, HRI can give benefits also in health care, for example in the rehabilitation domain or in the care of patients [37]. HRI is also used in autonomous guidance and space exploration.

2.4 Universal Robots

Universal Robots was founded in 2005 at Odense (Denmark) by three engineers, Esben Østergaard, Kasper Støy, and Kristian Kassow, who believed that the robotic market needed to become accessible also to small and medium sized businesses.

In fact, Universal Robot produces collaborative robot arms which can be used in a wide variety of applications. Thanks to six degrees of freedom, flexibility and easy integration into production environment, they are used in countless industries such as automotive, electronics and pharma.

Features	**UR3**	**UR5**	**UR10**	**UR16**
Reach	500 mm	850 mm	1300 mm	900 mm
Payload	3 kg	5 kg	10 kg	16 kg
Footprint	⌀128 mm	⌀149 mm	⌀190 mm	⌀190 mm
Weight	11.2 kg	20.6 kg	33.5 kg	31.1 kg

Table 2.2: Comparison table of features of different types of Universal Robots arms. Source:[2]

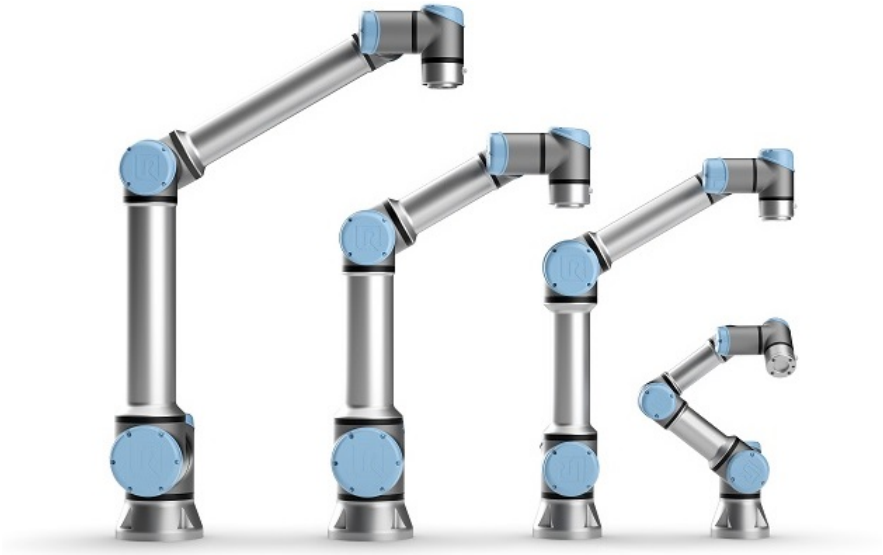

Figure 2.6: Four different models of Universal Robots arms

According to payload, as Table 2.2 shows, four different types of robotic arms (see Figure 2.6) are offered [2].

In Table 2.2, robotic arms are compared considering the following features:

- **Reach:** it is the distance between the center of the base of the robot and the point in which the robotic arm reaches its fullest extension;
- **Payload:** it is the maximum weight which the robotic arm can move;
- **Footprint:** it is the space occupied by the base of the robotic arm;
- **Weight:** it is the weight of the robotic arm.

Analyzing in depth the UR arm structure, it is a 6 axis robot arm and all the 6 axis can rotate by 360 degrees. The six joints, which are shown in Figure 2.7, are:

- **Base:** it is the joint between the platform to which the robot is attached and other components of the robot;
- **Shoulder:** it is the second axis of the robot and it is called in this way because it performs the same movement of the human shoulder;
- **Elbow:** it is the third axis of the robot and the name is due to the similarity with the movement performed by the human elbow;
- **Wrist 1:** it performs a bending;

- **Wrist 2:** it performs a rotation;
- **Wrist 3:** it performs a rotation of the end effector.

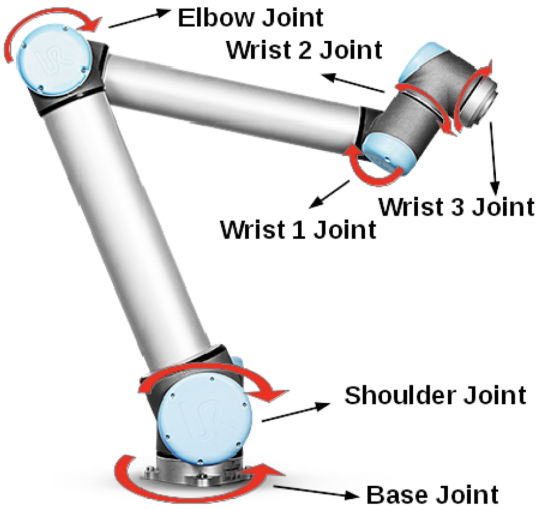

Figure 2.7: Universal Robot joints representation. Source:[50]

Universal Robots is made flexible and competitive with respect to other company thank to *UR+* which is a set of components and application kits compatible with UR arms. An example of components which could be integrated into robotic arm are vision systems, such as 2D and 3D cameras, different types of grippers (vacuum, electric, pneumatic), specific process end-effector able to weld, sand or dispense and software tools, such as simulators [2].

UR5

The UR5 is the robotic arm chosen for the three cases of study developed for this thesis due its size, which makes it particularly convenient (see Figure 2.9). As written before, the maximum payload is 5 kg although this value decreases with increasing distance between the end-effector and the robot center of gravity. The reachable workspace extends 850 mm from the base joint, as Figure 2.8 shows. The materials the robot is made of are aluminium and polypropylene. The control box is equipped with 16 digital input, 16 digital output, 2 analog input and 2 analog output ports. The tool is also equipped with 2 digital input, 2 digital output and 2 analog ports.

A complete overview technical specification about UR5 is in the Table 2.3.

Direct kinematics of UR5

Direct kinematics deals with the computing of end-effector pose as a function of the joint variables [47]. In an open-chain manipulator as UR5, which is constituted by 6 joints and 7 links, calling Link 0 the link which is fixed to the ground, the coordinate transformation which describes the position and the orientation of frame 6 with respect to frame 0 is:

$$\boldsymbol{T}_6^0(\boldsymbol{q}) = \boldsymbol{A}_1^0(q_1)\boldsymbol{A}_2^1(q_2)\boldsymbol{A}_3^2(q_3)\boldsymbol{A}_4^3(q_4)\boldsymbol{A}_5^4(q_5)\boldsymbol{A}_6^5(q_6) \tag{2.13}$$

where $\boldsymbol{q}$ is the vector of joint variables. To compute the direct kinematics equation, the Denavit-Hartenberg convention is generally used to define the relative position and orientation of two consecutive links. The Denavit-Hartenberg parameters suggests from UR website are shown in the Figure 2.10 listed in the Table 2.4.

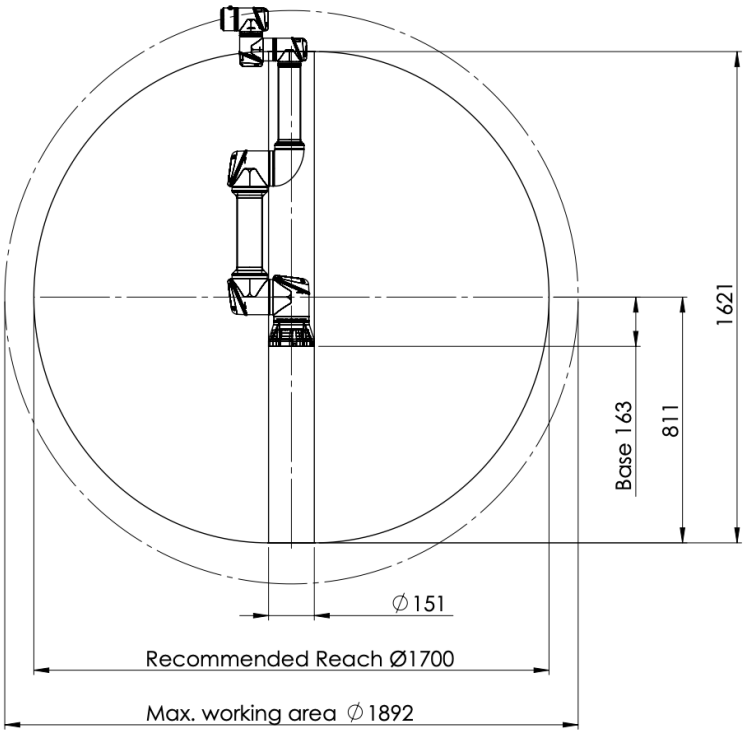

Figure 2.8: UR5 working area, side view. Source:[2].

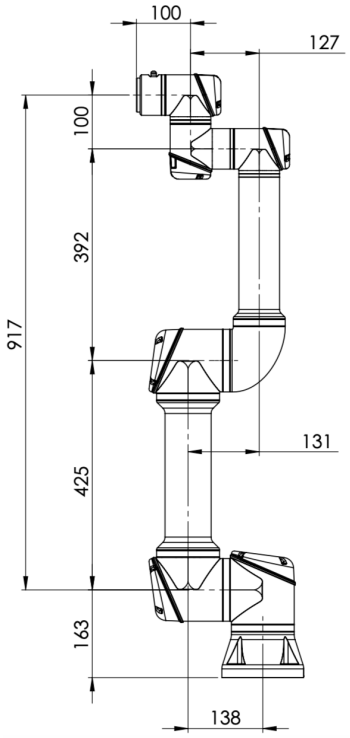

Figure 2.9: UR5 technical drawing. Source:[2].

Weight	20.7 kg
Maximum payload	5 kg
Reach	850 mm
Joint ranges	±360°
Speed	Joints: max 180°/s
System update frequency	500 Hz
Force Torque Sensor Accuracy	4 N
Pose Repeatability	± 0.03 according to ISO 9283
Footprint	⌀149 mm
Control Box Size (W × H × D)	460 mm × 449 mm × 254 mm

Table 2.3: Technical specifications of UR5. Source:[2]

The meanings of the Denavit–Hartenberg parameters are the following:

- $\boldsymbol{a_i}$ **:** is the distance between O_i and $O_{i'}$;
- $\boldsymbol{d_i}$ **:** is the coordinate of $O_{i'}$ along z_{i-1};
- $\boldsymbol{\alpha_i}$ **:** is the angle between axes z_{i-1} and z_i about x_i which is positive in counter-clockwise;

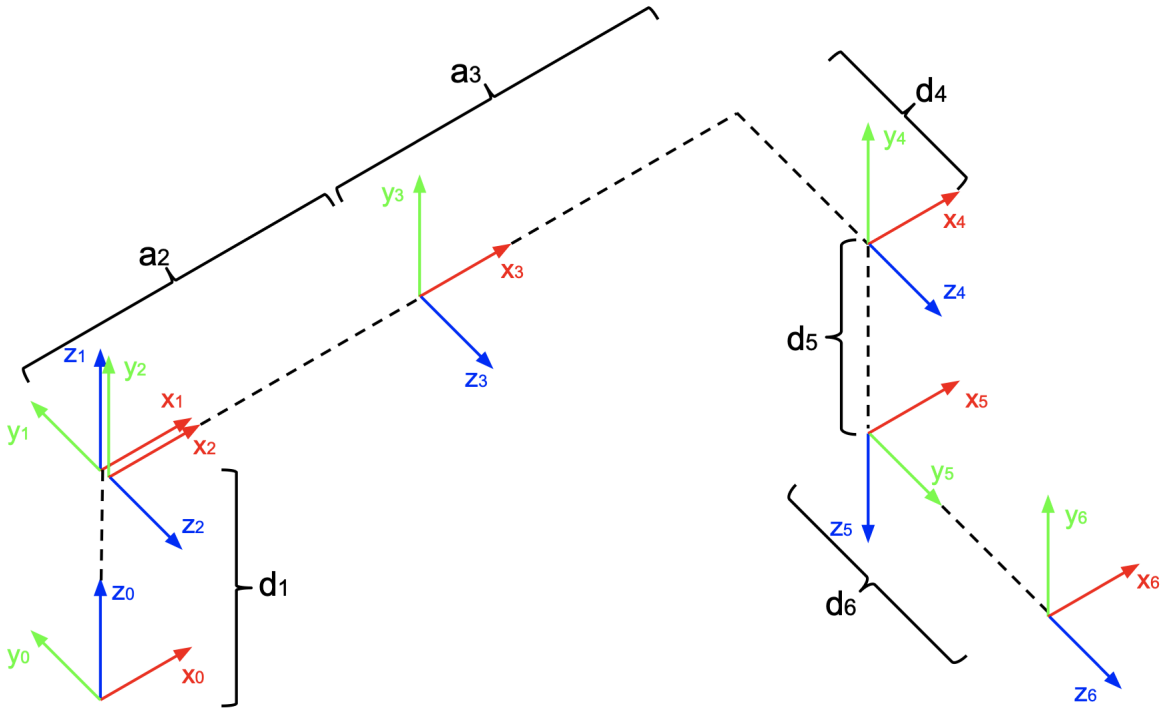

Figure 2.10: Denavit–Hartenberg parameters of Universal Robots arms. Source:[21]

Joints/Parameters	$\boldsymbol{\theta}$ [rad]	$\boldsymbol{a}$ [m]	$\boldsymbol{d}$ [m]	$\boldsymbol{\alpha}$ [rad]
Joint 0	–	0	–	0
Joint 1	θ_1	0	d_1	$\pi/2$
Joint 2	θ_2	a_2	0	0
Joint 3	θ_3	a_3	0	0
Joint 4	θ_4	0	d_4	$\pi/ 2$
Joint 5	θ_5	0	d_5	-π /2
Joint 6	θ_6	–	d_6	–

Table 2.4: List of Denavit–Hartenberg parameters of UR5.

- $\boldsymbol{\theta_i}$ **:** is the angle between axes x_{i-1} and x_i about z_i which is positive in counter-clockwise;

Chosen the Denavit–Hartenberg parameters, using the Denavit–Hartenberg convention allows to construct the direct kinematic function concatenating the individual coordinate transformations whose generalization is written in the following equation:

$$\boldsymbol{A}_i^{i-1}(q_i) = \boldsymbol{A}_{i'}^{i-1}\ \boldsymbol{A}_i^{i'} = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

Thus, solving the equation 2.13 it is possible to find the solution of the direct kinematic problem

[21]:

$$\boldsymbol{T}_6^0(\boldsymbol{q}) = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

where

$$n_x = c_6(s_1 s_5 + ((c_1 c_{234} - s_1 s_{234})c_5)/2 + ((c_1 c_{234} + s_1 s_{234})c_5)/2 - s_6((s_1 c_{234} + c_1 s_{234}) - (s_1 c_{234} - c_1 s_{234})))/2 \tag{2.16}$$

$$n_y = c_6(((s_1 c_{234} + c_1 s_{234})c_5)/2 - c_1 s_5 + ((s_1 c_{234} - c_1 s_{234})c_5)/2) + s_6((c_1 c_{234} - s_1 s_{234})/2 - (c_1 c_{234} - s_1 s_{234})/2) \tag{2.17}$$

$$n_z = (s_{234} c_6 + c_{234} s_6)/2 + s_{234} c_5 c_6 - (s_{234} c_6 - c_{234} s_6)/2 \tag{2.18}$$

$$o_x = -(c_6((s_1 c_{234} + c_1 s_{234}) - (s_1 c_{234} - c_1 s_{234})))/2 - s_6(s_1 s_5 + ((c_1 c_{234} - s_1 s_{234})c_5)/2 + ((c_1 c_{234} + s_1 s_{234})c_5)/2) \tag{2.19}$$

$$o_y = c_6((c_1 c_{234} - s_1 s_{234})/2 - (c_1 c_{234} + s_1 s_{234})/2) - s_6(((s_1 c_{234} + c_1 s_{234})c_5)/2 - c_1 s_5 + ((s_1 c_{234} - c_1 s_{234})c_5)/2) \tag{2.20}$$

$$(c_{234} c_6 + s_{234} s_6)/2 + (c_{234} c_6 - s_{234} s_6)/2 - s_{234} c_5 s_6 \tag{2.21}$$

$$a_x = c_5 s_1 - ((c_1 c_{234} - s_1 s_{234})s_5)/2 - ((c_1 c_{234} + s_1 s_{234})s_5)/2 \tag{2.22}$$

$$a_y = -c_1 c_5 - ((s_1 c_{234} + c_1 s_{234})s_5)/2 + ((c_1 s_{234} - s_1 c_{234})s_5)/2 \tag{2.23}$$

$$a_z = (c_{234} c_5 - s_{234} s_5)/2 - (c_{234} c_5 + s_{234} s_5)/2 \tag{2.24}$$

$$p_x = -(d_5(s_1 c_{234} - c_1 s_{234}))/2 + (d_5(s_1 c_{234} + c_1 s_{234}))/2 + d_4 s_1 - (d_6(c_1 c_{234} - s_1 s_{234})s_5)/5 - (d_6(c_1 c_{234} + s_1 s_{234})s_5)/2 + a_2 c_1 c_2 + d_6 c_5 s_1 + a_3 c_1 c_2 c_3 - a_3 c_1 s_2 s_3 \tag{2.25}$$

$$p_y = -(d_5(c_1 c_{234} - s_1 s_{234}))/2 + (d_5(c_1 c_{234} + s_1 s_{234}))/2 - d_4 c_1 - (d_6(s_1 c_{234} + c_1 s_{234})s_5)/2 - (d_6(s_1 c_{234} - c_1 s_{234})s_5)/2 - d_6 c_1 c_5 + a_2 c_2 s_1 + a_3 c_2 c_3 s_1 - a_3 s_1 s_2 s_3 \tag{2.26}$$

$$p_z = d_1 + (d_6(c_{234} c_5 - s_{234} s_5))/2 + a_3(s_2 c_3 + c_2 s_3) + a_2 s_2 - (d_6(c_{234} c_5 + s_{234} s_5))/2 - d_5 c_{234} \tag{2.27}$$

Inverse kinematics of UR5

The determination of the joint variables which corresponds to a given pose of the end-effector is called inverse kinematic problem. Solving this problem is fundamental to transform the motion assigned to the end-effector in the operational space into the corresponding joint space motions and thus solving this problem allows execution of the desired displacement of the end-effector [47]. Thus, computing the inverse kinematic of a UR5 robotic arm means computing the joint angles $[\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]$ which satisfy equation 2.28.

The desired transformation matrix referred to the reference frame 6 can be also written as [52]:

$$\boldsymbol{T}_6^0(\boldsymbol{q}) = \begin{bmatrix} n_6x & o_6x & a_6x & p_6x \\ n_6y & o_6y & a_6y & p_6y \\ n_6z & o_6z & a_6z & p_6z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_6 & \boldsymbol{p}_6 \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{2.28}$$

Considering the transformation between frame 1 and 5 [21]:

$$(\boldsymbol{A}_1^0)^{-1}\ \boldsymbol{T}_6^0\ (\boldsymbol{A}_6^5)^{-1} = \boldsymbol{A}_2^1\ \boldsymbol{A}_3^2\ \boldsymbol{A}_4^3\ \boldsymbol{A}_5^4 \tag{2.29}$$

where $\boldsymbol{T}_6^0$ describes the desired position and orientation of the end-effector link.

As the Figure 2.10 shows, $\boldsymbol{p}_5$ can be determined by moving $\boldsymbol{p}_5$ along z axis of a distance $-d_6$.

$$\boldsymbol{p}_5 = \begin{bmatrix} p_{5x} \\ p_{5y} \\ p_{5z} \end{bmatrix} = \boldsymbol{p}_6 - d_6 \begin{bmatrix} a_{5x} \\ a_{5y} \\ a_{5z} \end{bmatrix} \tag{2.30}$$

Knowing $\boldsymbol{p}_5$, θ_1 can be computed as:

$$\theta_1 = atan2(p_{5y}, p_{5x}) \pm cos^{-1} \frac{d_4}{\sqrt{p_{5x}^2 + p_{5y}^2}} + \frac{\pi}{2} \tag{2.31}$$

The two solutions of the equation 2.31 correspond to the left and right position of the shoulder.

Knowing θ_1, θ_5 and θ_6 are determined as:

$$\theta_5 = \pm cos^{-1} \left(\frac{p_{6x}s\theta_1 - p_{6y}c\theta_1 - d_4}{d_6} \right) \tag{2.32}$$

$$\theta_6 = atan2 \left(\frac{-o_{6x}s\theta_1 + o_{6y}c\theta_1}{s\theta_5}, \frac{n_{6x}s\theta_1 - n_{6y}c\theta_1}{s\theta_5} \right) \tag{2.33}$$

The other 3 joints can be treated as a classical 3 links planar arm, where the reference frame 2 represent the base and reference frame 4 represents the end-effector.

Knowing θ_1, the homogeneous transformation matrix between the reference frame 0 and 1 can be written according to equation 2.14:

$$\boldsymbol{T}_1^0 = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_ic_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_is_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_1 & \boldsymbol{p}_1 \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{2.34}$$

The reference system 3 cannot be defined yet due to the unknown joint angles between the base and the end-effector of the three links planar arm.

Since θ_5 and θ_6 are known, it is possible compute the homogeneous transformation matrix between frames 4 and 6:

$$\boldsymbol{T}_4^0 = \boldsymbol{T}_6^0\ (\boldsymbol{T}_6^4)^{-1} = \begin{bmatrix} \boldsymbol{R}_4 & \boldsymbol{p}_4 \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{2.35}$$

Knowing the position vector $\boldsymbol{p}_4$, it is possible compute $\boldsymbol{p}_3$ by moving along the reference system 4 along the y axis by $-d_4$:

$$\boldsymbol{p}_3 = \begin{bmatrix} p_{3x} \\ p_{3y} \\ p_{3z} \end{bmatrix} = \boldsymbol{p}_4 - d_4 \begin{bmatrix} o_{6x} \\ o_{6y} \\ o_{6z} \end{bmatrix} \tag{2.36}$$

Since all the remaining position vectors are known, the joint angles θ_2, θ_3 and θ_4 can be computed according to the following equation [47]:

$$\theta_3 = \pm cos^{-1}\left(\frac{p_{4x}^2 + p_{4y}^2 - a_2^2 - a_3^2}{2a_2a_3}\right) \tag{2.37}$$

$$\theta_2 = atan2(p_{4y}, p_{4y}) \pm cos^{-1}\left(\frac{p_{4x}^2 + p_{4y}^2 - a_2^2 - a_3^2}{2a_2\sqrt{p_{4x}^2 + p_{4y}^2}}\right) \tag{2.38}$$

$$\theta_4 = \psi - \theta_2 - \theta_3 \tag{2.39}$$

where ψ is the angle between joint 4 and the x_2 axis. According to the sign of θ_3, the elbow can be in up or down posture.

2.4.1 Universal Robots grippers - Robotiq 2F-85

As written before, Universal Robot provides a framework, called UR+, which includes different kinds of grippers compatible with UR arms. The gripper chosen for the second and third case of study is 2F-85 of Robotiq. The 2F stays for 2 Finger and 85 is referred to the finger opening dimension, which is 85 mm, as Figure 2.11 shows. It is suitable for different kinds of application, such as material handling, machine tending, assembly and quality inspection. It allows to pick, place and handle a large range of objects of varying sizes and shapes. This gripper is particularly suited for the last two cases of study since it has five points of contact with an object and it can adapt to the shape of the object they grasp [45].

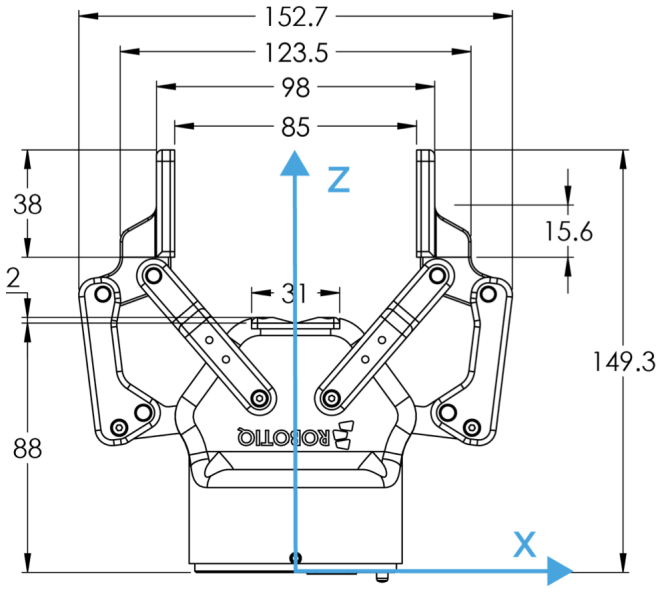

Figure 2.11: Technical drawing of Robotic 2F-85. Source:[45].

2.4.2 Universal Robots communication interfaces

Communication between Universal Robots arms and external devices can take place using different types of communication interfaces, as Figure 2.12 shows:

- **Primary/Secondary interfaces:** in the primary interface the controller sends robot state data and additional messages and receives URScript commands, while in the secondary interface the controller transmits only robot state data and receives URScript commands. In both interfaces, URScript commands have an update rate of 10 Hz;
- **Real-time interfaces:** the controller transmits the robot state data and receives URScript commands with an update rate of 500 Hz;
- **Dashboard server:** using a TCP/IP socket, an Universal Robots arm can be controlled remotely sending to the GUI simple commands, such as load, play, pause and stop a robot program and receive feedback about the state of the robot;
- **Socket communication:** data can be transferred between robot and external devices through TCP/IP protocol. In this type of communication, the robot is the client and the external device is the server;
- **XML-RPC:** it is a Remote Procedure Call method which transfers data using XML. The controller can call methods and functions on a server or remote program and it receives structured data. URScript cannot perform this type of calculation, but this interface is available for common programming languages such as Python, C, C++ and Java;
- **Real-Time Data Exchange (RTDE):** it synchronizes external applications with the UR controller using a TCP/IP connection. It allows the interaction with fieldbus drivers, the manipulation of input and output of the robot and the plot of the robot status [2].

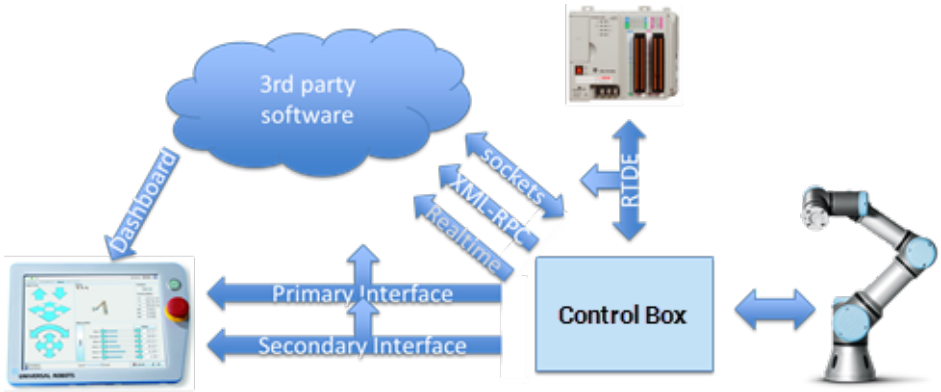

Figure 2.12: Different types of interfaces of UR. Source: [2]

The main features of each type of interface are summarized in the Table 2.5.

Real-Time Data Exchange - RTDE

In this thesis, the interface selected for all three cases of study developed in this thesis is the Real-Time Data Exchange, since this type of interface is robust for real-time applications and it does not break any real-time properties of the controller. A significant advantage of using this interface is that the synchronization is configurable: for example, the output might consist in tool, joint status or general purpose output register and the input might consists in general purpose input register or D/A outputs. Using RTDE, the client is responsible for the setup of

Features/ Interfaces	**Primary interface**	**Secondary interface**	**Real-time interface**	**XML-RPC**	**RTDE**
Port number	30001/30011	30002/30012	30003/30013	Not used occupied or frequently used port	30004
Frequency	10 Hz	10 Hz	500 Hz	–	500 Hz
Transmit	Robot state data and messages	Robot state data	Robot state data	Call methods and functions	Diverse types of data
Receive	URScript commands	URScript commands	URScript commands	Structured data	Diverse types of data

Table 2.5: Comparison table of features of most common used UR interfaces. Source:[2]

the variables to be synchronized. To implement the RTDE client different languages can be used. In this case Python is chosen, since Universal Robots provides a RTDE client library implemented using this programming language [2].

To implement this kind of interface, in the main program it is necessary to specify:

- **Robot host:** it is the IP address to connect to;
- **Port number:** as previously mentioned, for RTDE the port is 30004 by default;
- **A command to read the configuration file:** a configuration file is an xml file which contains the recipes. A recipe is the definition of a synchronization data package and contains the name of the variable to synchronize and their types (integer, float, boolean);
- **Start data synchronization:** if no problem occurs, the connections is established;
- **Control loop:** after the starting of the synchronization loop, the RTDE sends the client the requested data in the same order it was requested by the client.

Chapter 3

Tools overview

This chapter is intended as an overview of both the simulation environments and the library used to implement the computer vision algorithms required for the applications developed in this thesis.

3.1 Simulation Environments

Simulators play a fundamental role in testing in an efficient and quick way the behavior of robots in various working conditions. Offline simulators allow to design in a proper way robotized stations, while online simulators allow to have a feedback between the software and the robot. Two main types of simulators exist:

- **Proprietary software:** it is provided by the robot manufacturers and it is used for programming and testing robots offline in a 3D environment. Motion trajectory and control programs are created using programming languages which are compatible with programming languages of real robots.
- **Universal programs:** they are programs for simulating and analyzing the behavior of the robot in a certain environment. They allow to compare the performances of different robots in the same environment. The programming language is a standard programming language, such as Python, Matlab, C, depending on the simulation software. The program can be compiled and translated to basic robot programming languages [27].

3.1.1 Polyscope

Polyscope is a graphic programming environment provided by Universal Robots which allows the user to program the robot using the *teach pendent* of the robot or a computer (see Figure 3.1). The teach pendent is a hand-held device which the operator can use to write a program in a quick and easy manner and to send the program to the robot. Using Polyscope on a computer allows to simulate the behavior of the robot to test it or to run directly the program on the robot, without using a teach pendent. Polyscope simplifies the work of the operator who can use this easy graphical programming language instead of programming the robot using *URScript*, the proprietary programming language of Universal Robots.

Polyscope allows the user to program the robot setting a "before start" sequence, waypoints and home position and performing different types of movement. In particular, three different types of movements might be set:

- **MoveJ:** it performs a linear movement in joint-space. This move allows to reach speeds higher than the other two movements;

- **MoveL:** it performs a linear movement in tool-space. Generally, it is used in pick and place applications when the robot approaches the object to pick and when the robot drops the object and moves away.
- **MoveP:** it performs a process movement. It is a blend of circular and linear movement. The movement happens at constant speed.

Polyscope enables to program basic flow control capabilities such as if statements, loops and threads.

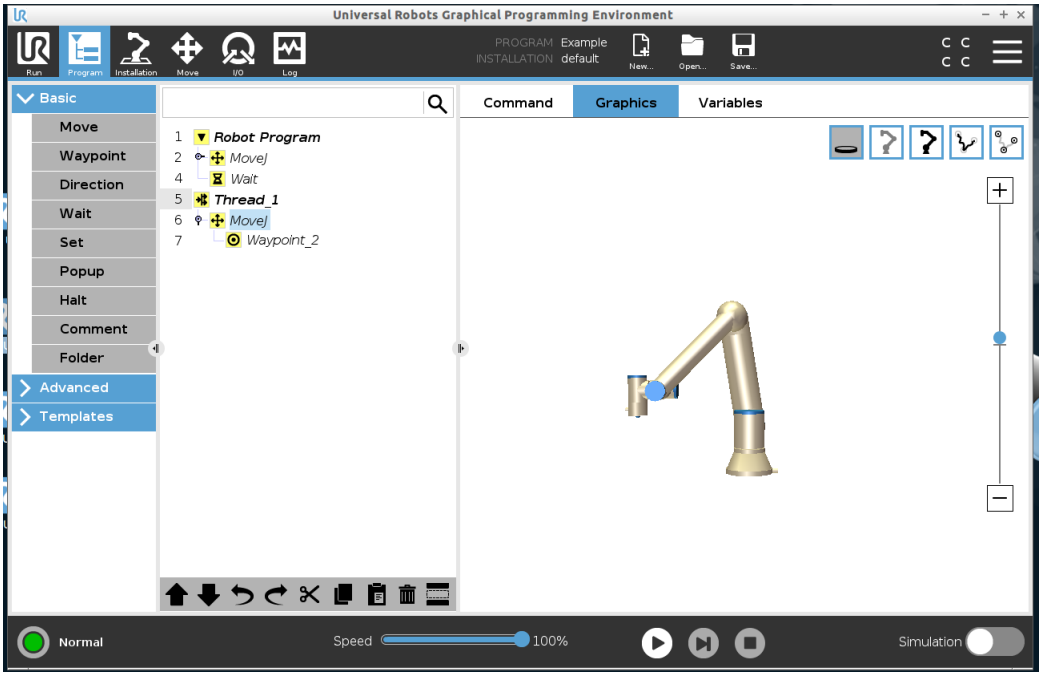

Figure 3.1: The Polyscope programming and simulation environment

In Polyscope it is also possible to add *URCaps*, which are accessories and hardware that are useful to the robot to perform a given task. In this way, actions such as opening and closing the gripper can be added directly in this graphical programming environment.

Moreover, Polyscope allows to set the desired safety configuration such as defining a safety area and setting a safety mode choosing between the three following options:

- **Reduce mode:** the movement of the robot is slower when a person is nearby it;
- **Safeguard stop and continue:** the robot stops when a person is nearby it;
- **Safeguard stop and reset:** the robot stops when a person is nearby it and it is resumed pressing the restart button.

3.1.2 RoboDK

RoboDK is a simulator for industrial robots and offline programming born in 2015 from a spin-off company of ETS University Montreal (Canada). Nowadays, its library consists of over 500 robot arms and tens of different types of grippers and end-effector.

It allows to simulate and to offline program robots in an easier manner, without learning the proprietary programming language of the robot and with the capability of exporting programs to the robot.

It is also possible to import 3D objects in different formats such as STL, STEP, IGES, WRML, 3DS or OBJ. In this way, the behavior of the robot is simulated in a realistic environment (see Figure 3.2).

Robot positions can be saved as targets. Two types of target exist: a cartesian target defines the position of the tool with respect to a coordinate system, while a joint target defines the position of the robot given robot joint values. A target can also be created on a surface: this type of target is useful for applications such as inspection and painting [44].

Another interesting feature of RoboDK is the possibility to add a Python script and perform actions such as closing or opening a gripper, attaching and detaching objects.

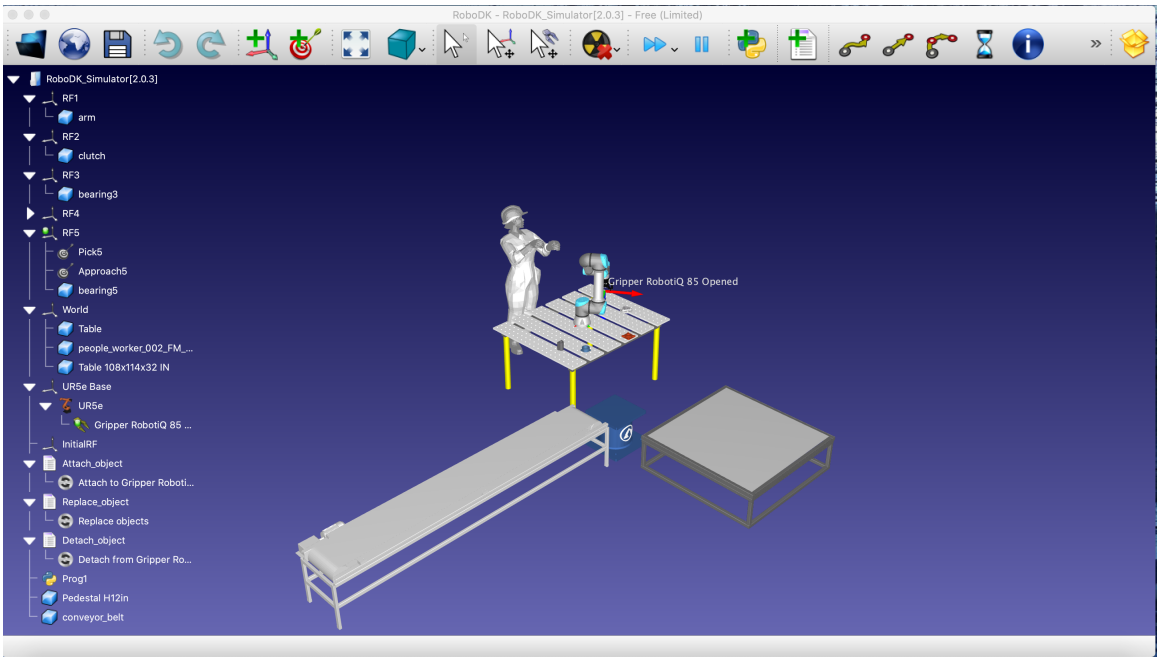

Figure 3.2: RoboDK simulation environment with imported 3D objects.

3.2 OpenCV

OpenCV is an open source computer vision library. It is written in *C* and *C++* and has interfaces for *Python*, *Matlab*, *Java* and others programming languages. It is mainly aimed at real-time computer vision. The OpenCV developers' goal is to provide the user with a simple computer vision infrastructure with which the user can quickly realize sophisticated computer vision applications. Since the importance of machine learning in several computer vision applications, OpenCV provides also a general-purpose machine learning library focused on clustering and statistical pattern recognition.

The idea beyond OpenCV was born from an Intel researcher who, while visiting some prestigious university groups, such as the MIT Media Labs, noticed that an open computer vision infrastructure was internally developed and the code passed from student to student and, in this way, each new student had a good infrastructure from which he could start to develop his own computer vision application. Thus, with the aid of Intel's Performance Library Team, OpenCV codes and algorithms started to be implemented [8].

Several applications can be performed using the OpenCV library, such as face and object recognition, automated surveillance and inspection, driver-less car and robot control, medical image analysis. The main functionalities of OpenCV consists in image and video processing and display, feature detection, machine learning and clustering. In Chapter 4 the theory beyond several computer vision function available on OpenCV is analyzed.

Chapter 4

Computer Vision techniques

Computer Vision is a term used to refer to the science and technology of extracting relevant data from digital images or videos, as well as the large number of techniques and algorithms developed to achieve this.

Computer Vision encompasses a wide variety of sub-fields, each with its own objective, which may or may not be related to each other.

In this chapter, a rough taxonomy of the various sub-fields of Computer Vision is presented at first. Then, after identifying those that are relevant for the applications developed for this thesis, the theory behind the techniques that were used for the applications presented in Chapter 5 is outlined.

4.1 The taxonomy of Computer Vision

The applications of Computer Vision are manyfold. Some of the tasks that are currently widely used in industrial and user applications include:

- **Object recognition:** classify an image among a set of pre-defined classes or labels.
- **Object identification:** determine whether a specific object is present in an image, and where.
- **Object detection:** determine if any object belonging to a set of pre-defined classes is present in an image and localize it/them.
- **Object segmentation:** partition an image into its separate objects.

Often, these tasks are achieved with the integration of Machine Learning (ML) and Deep Learning techniques, such as *Convolutional Neural Networks* (CNN). These allow to extract information from images at a level of abstractions there was previously not possible. Despite their drawbacks, such as the requirement for an appropriate *training* phase with large amount of input data, their unmatched performance has proven to be key to enable performing the high-level tasks previously listed in practical applications.

However, one notable weakness of AI techniques is that they are extremely sensitive to the quality of the input data. In practical terms, if the image that is fed to the ML model (such as a CNN) does not have a structure that is adequately close to the images that were used to train it, the model output will strongly differ from what we would expect. This concept is very popular in the AI world by the catchphrase *"Garbage in, garbage out"*.

Therefore, a great deal of effort is put in pre-processing the images that are acquired by the acquisition system (e.g. a camera) so that they are easy to process with any ML model,

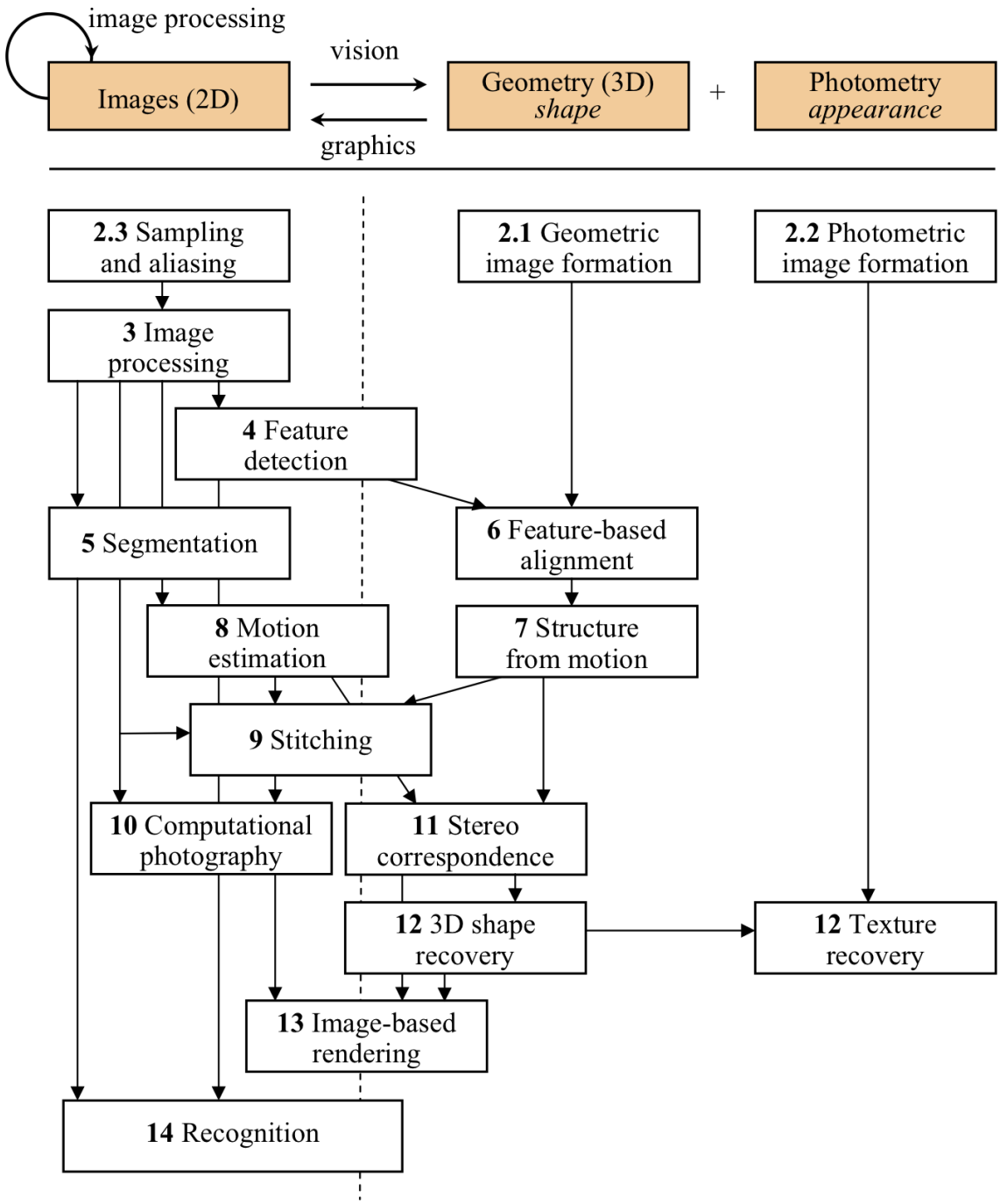

Figure 4.1: Taxonomy of Computer Vision fields. Source: [49].

allowing it to provide a meaningful output. These pre-processing techniques, such as image filtering and segmentation, are also encompassed in the family of Computer Vision techniques.

An interesting and clear classification of the various fields that are included in Computer Vision can be found in [49] and is schematized in Figure 4.1.

The remaining part of this chapter deals with the theory and techniques that are relevant for the applications developed in this thesis. First, Section 4.2 introduces the way in which digital images are represented. Then, Section 4.3 illustrates some image processing techniques, such as thresholding (which is used to achieve image segmentation), filtering, and morphological transformations. Finally, Section 4.4 deals with the techniques used for object detection. In particular, Section 4.4.1 and 4.4.2 introduce the use of deep learning models for object recognition and detection, while Section 4.4.3 illustrates the use of geometrical features that were

used to finally achieve gesture recognition.

4.2 Digital images representation

4.2.1 Gray-scale images

In general, cameras acquire images by using image sensors which are able to measure the amount of incident light during a certain *exposure* time [35]. Each sensor captures the information that define a *pixel* and the image itself is made of a rectangular array of pixels (see Figure 4.2).

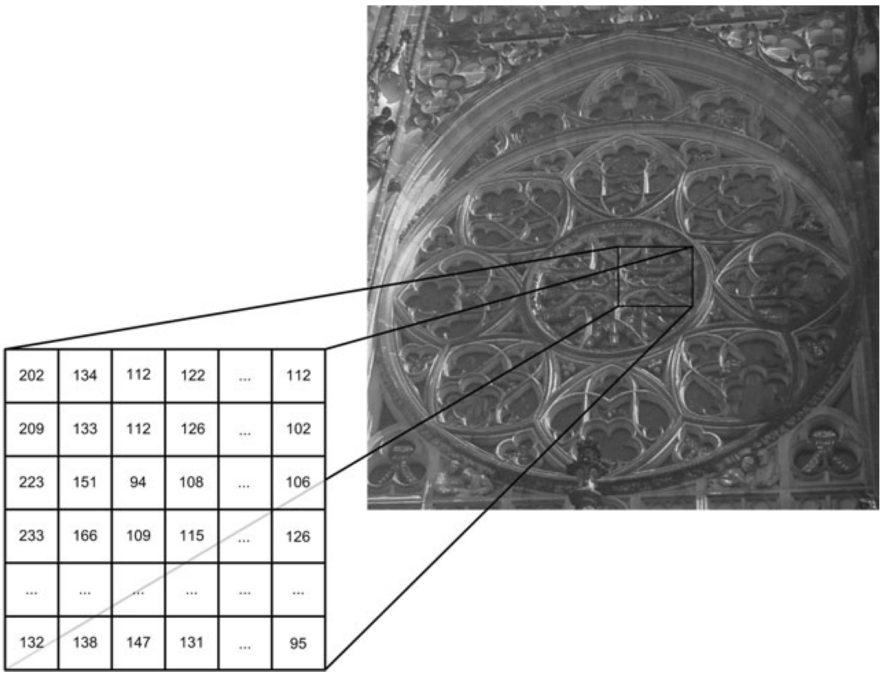

Figure 4.2: A gray-scale image with pixel intensity values for a small portion of it. Source: [35].

This information is then digitized for each sensor, usually as an 8-bit value, which means that 256 different levels of light intensity can be represented. The intensity of each pixel is visualized as a shade of gray, with the highest value (255) being associated to pure white and the lowest (0) to pure black.

4.2.2 Color images and color spaces

To capture color as well as light intensity, cameras acquire color images by using three sensors for each pixel, which measure the amount of red, green and blue light [35]. This reflects the way in which humans build the concept of color.

The human eye contains a large number of photoreceptors called *cones*, which are divided in three type each of which is sensible to a particular wavelength range, and our brain constructs the colors we perceive based on the intensities sensed by our cones. These three ranges correspond to what we call the three *additive* primary colors (red, green and blue).

Therefore, color images produced by digital cameras are composed by pixels each of which is characterized by three color intensities (the RGB values). Typically, each color value is stored in an 8-bit value, which means that 256 shades of each color can be represented.

Therefore, a digital image can be represented by splitting it into three different images, each representing the intensity of the associated color (or more generally *channel*), as can be seen in Figure 4.3.

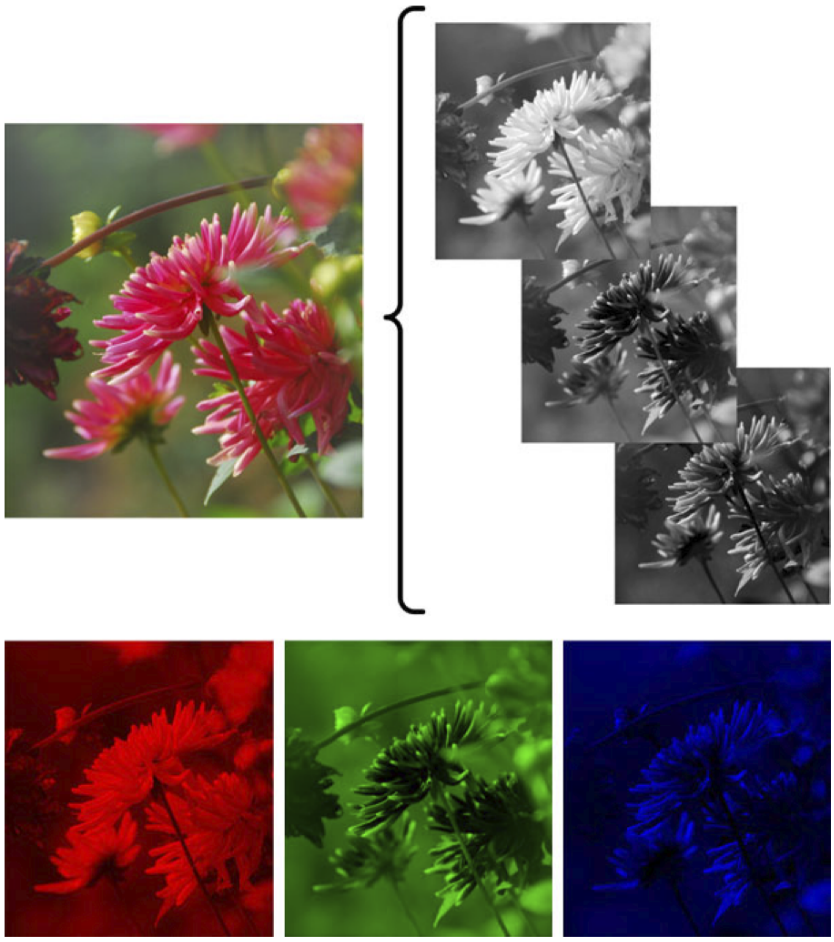

Figure 4.3: Decomposition of an image into its RGB channels. Source: [35].

4.2.3 The YCbCr color space

Although RGB is the primary color space for encoding digital images, several other color spaces are used in image processing, such as *HSV* or *YCbCr*. YCbCr is a color space that has historically been used for video transmission, but has recently proven useful in other applications as well. The first channel, Y, stands for luminance, while the other two are the blue-difference and red-difference *chroma* components. The *chroma* or *chrominance* channels are directly related to the intensity differences $B-Y$ (blue-luminance difference) and $R-Y$ (red-luminance difference).

For the purposes of the applications presented in this thesis, the most interesting aspect of YCrCb is that is effectively separates the luminance from the chromatic aspect of the color of the image. This makes it quite useful for skin recognition.

In fact, illumination conditions strongly influence object recognition tasks where the skin color is relevant, for instance face or hand recognition. For this reason, it can be useful to adopt the YCrCb color space and only consider the two chromatic channels [31].

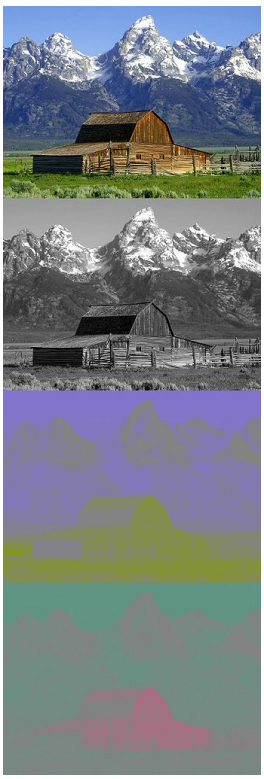

Figure 4.4: Decomposition of an image into its YCrCb channels. Source: [54].

4.3 Image processing techniques

4.3.1 Thresholding

Thresholding is a simple operation that attempts to separate an object from its background. As such, it can be used to achieve image segmentation.

Thresholding is the most common point processing operations. Point processing operations, as opposed to neighborhood operations, are simply operations that operate pixel-wise, and the pixels of the resulting image are not affected by the values of the neighboring pixels.

The process of thresholding converts a gray-scale image to a binary image, where all of the pixels' intensities become either 0 (black) if they are below or 255 (white) if they are over a certain threshold (see Figure 4.5). In mathematical terms [49]:

$$g(x,y) = \begin{cases} 0 & \text{if } f(x,y) \leq t \\ 255 & \text{if } f(x,y) > t \end{cases} \tag{4.1}$$

where f is the original image, g is the thresholded image, and t is the threshold.

The threshold can be either set manually or with some selection algorithm such as Otsu's algorithm, which attempts to minimize the weighted within-class variance of the white and black classes [56].

4.3.2 Filtering and the Gaussian Blur

The *Gaussian Blur* is a popular *blurring* (or *smoothing*) filter used to filter out noise from the image.

In short, applying a filter to an image means that the intensity of each pixel of the resulting image is given by a weighted sum of the intensities of pixels of the original image in a surrounding

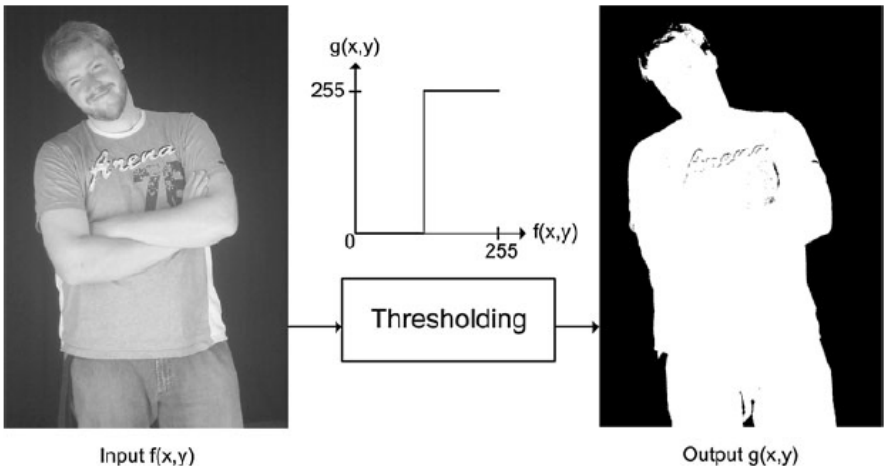

Figure 4.5: Example of thresholding on an image. Source: [35]

area (see Figure 4.6). More formally, filtering is a convolution operation between the original image and the filter [41].

Edge detection algorithms perform poorly with almost any real-life images, as they are unable to differentiate the edges that are most relevant from other non-interesting discontinuities. Sharp variations in intensity caused by noise can lead the edge detection algorithm to identify edges where there should be none. Similarly, thresholding noisy images is often ineffective in differentiating objects from their background.

For this reason, the image must be smoothed to suppress as much noise as possible.

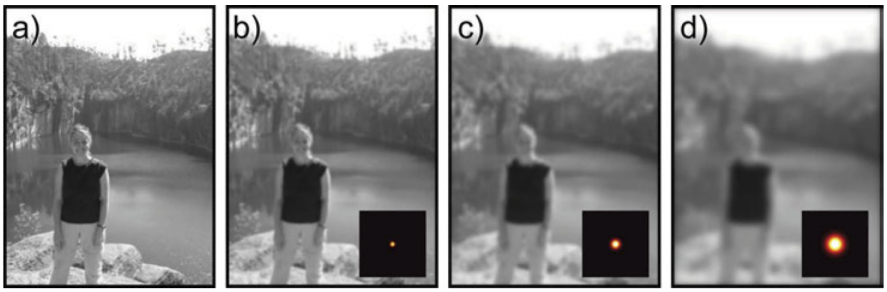

Figure 4.6: Example of Gaussian blur on an image with increasingly large standard deviation. Source: [41].

Gaussian Blur uses a Gaussian function as a filter. The formula for the two-dimensional Gaussian function is [41]:

$$f(x,y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}. \tag{4.2}$$

where x and y are the pixel coordinates, and σ is the standard deviation of the Gaussian. In practice, using a Gaussian filter means that nearer pixels will weight more in determining the result of the intensity of the pixels of the resulting image.

4.3.3 Morphological transformations

Morphological transformations mainly act on the image shape and can further reduce noise. Similar to filtering, they are neighborhood operators which apply a *kernel* or *structuring element* to the original image [35]. This kernel is basically a binary matrix, where the ones are typically arranged to form a square or a circle whose size determines the size of the neighboring region which will influence the pixels of the processed image.

However, differently from filtering, morphological transformations are not simply equivalent to the convolution of the kernel with the original image. Rather, the result of the transformation is determined based on the thresholded value of the convolution [49].

The most common basic morphological transformations are:

- dilation
- erosion
- majority

Dilation dilates the shape of the object, while *erosion* shrinks it.

Often, these transformations are used together in compounded transformations. *Opening* is erosion followed by dilation (see Figure 4.7 and 4.8), while *closing* is the opposite.

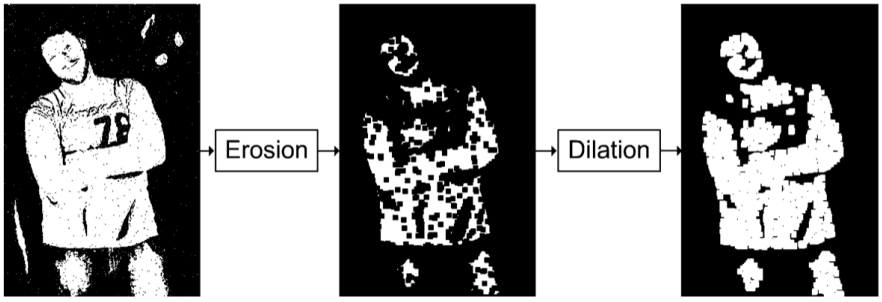

Figure 4.7: Example of opening operation on a binary image. Source: [35].

Figure 4.8: Another example of opening operation on a binary image. Source: [51].

Both these compounded transformations can be used to eliminate noise and obtained a simpler and cleaner topology for the objects which are relevant to use in our figure.

4.3.4 Identifying Contours

Contours are the boundary curves that can be drawn around individual objects of interest in an image. Drawing the contour of an object is one of the common objectives of image processing, and a prerequisite to many object recognition tasks.

Identifying the contours can be done in many different ways. One of the most common is to use an *Edge Detection* algorithm on our image to identify all edges, and then separating the non-relevant edges from the ones that define the relevant contours.

Edges are defined as lines where rapid intensity variation occurs [49], and may therefore include shadow boundaries, changes in surface orientations, variations in lighting, and many other undesired features.

An alternative to using an edge detection algorithm is to use thresholding. In fact, after a binary image is obtained, the task of finding the objects contours is trivial. In this case, the

quality of the obtained contour depends on the goodness of the thresholded image in separating the objects from the background.

This last approach in particular is the one that will be used in the applications developed for this thesis, as mentioned in Chapter 5.

4.4 Object Recognition and Detection

4.4.1 Image Recognition with Neural Networks

Neural Networks for Regression

Artificial Neural Networks are a family of Deep Learning models that are used to identify and "learn" patterns from large datasets.

A neural network in its simplest form is composed by:

- An *input* layer, which is essentially a real-valued vector which constitutes our input data.
- A series of *hidden* layers, which perform computations on the values coming from the preceding layers. Each hidden layer is composed by a certain number of *neurons* and each neuron has several coefficients, which are called *weights*, associated to it. Additionally, each layer also has a constant term called *bias*.
- An output layer, which produces a real-valued vector which characterizes our output data.

Such a simple neural network is also known as the *Multi-Layer Perceptron* architecture [16].

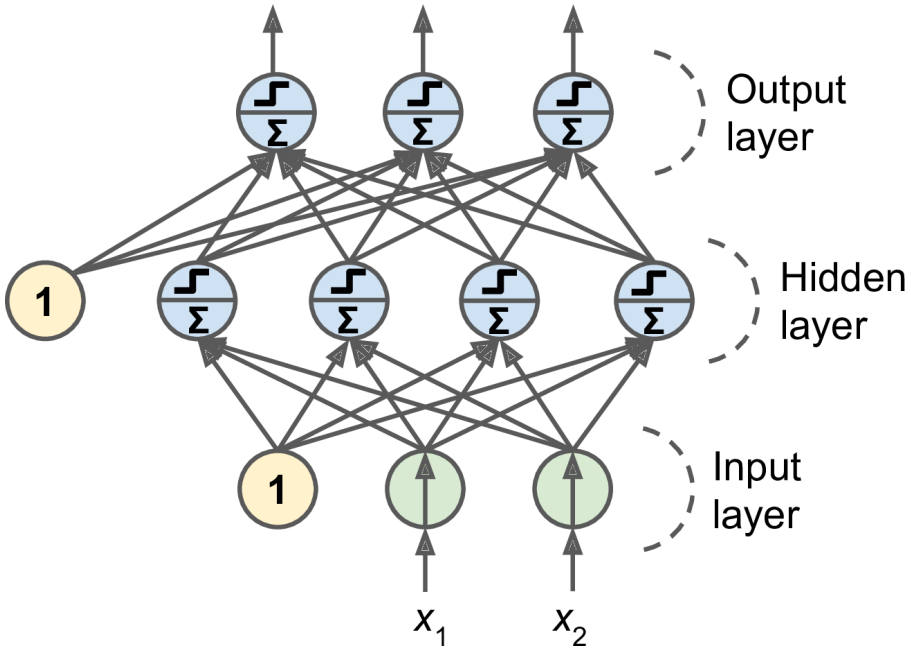

Figure 4.9: A regression neural network in its simplest form, with a two-dimensional input, a three-dimensional output and only one hidden layer. The bias terms are colored in yellow. Source: [16]

A scheme of a such a simple neural network architecture can be seen in Figure 4.9. Each neuron computes an output based on the inputs coming from all of the neurons of the preceding layer, and passes it to all the neurons of the following layer. Neurons compute their output in two steps: first, they compute a weighted sum of their inputs and the bias terms. Then, they pass this output through an activation layer.

The weighted sum performed by a neuron j can be written as:

$$y'_j = \sum w_{i,j} x_i + b_j, \tag{4.3}$$

where x_i are the inputs to the neuron j (the outputs of the preceding layer), $w_{i,j}$ are the weights associated to the connection between neuron j and neuron i of the preceding layer, and b_j is the bias term associated to the neuron.

Then, the output of the neuron is:

$$y_j = \phi(y'_j), \tag{4.4}$$

where ϕ is some activation function. This activation function allows to suppress or attenuate the neuron output if it does not overcome a certain threshold. The role of activation functions (see Figure 4.10) is to introduce non-linearities in the neural network, thus allowing it to reproduce any arbitrary (and, possibly, complex) non-linear function.

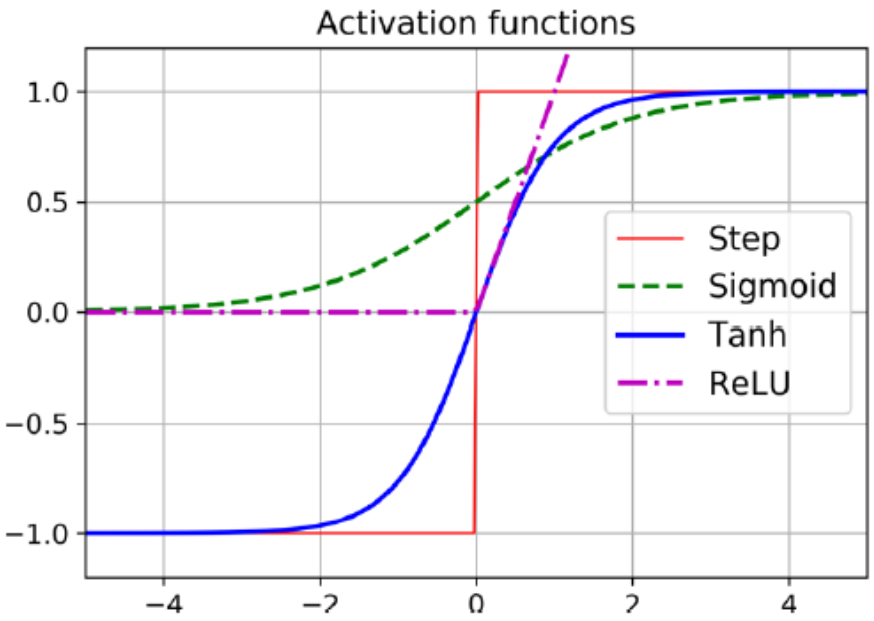

Figure 4.10: Some of the most popular activation functions. Source: [16]

The kind of neural network previously described can be trained on large datasets, where both inputs and the "true" outputs are known, and then used to perform predictions on the output of new input data, whose true output is unknown. In other words, they can be trained to perform regression tasks.

Neural Networks for Classification

While regression tasks deal with real-valued outputs, classification (as is the case for image recognition) deals with the case where the output is a label. To perform classification tasks, the neural network architecture that was previously described, a *Softmax* layer can be added (see Figure 4.11).

In this case, we have one output neuron for each of the possible classes our input data object can belong to, and we interpret their value as the probability that the object belongs to that particular class. In order to make sure that all probabilities are between 0 and 1 and they add up to 1 (so that we can indeed interpret them as probabilities), the Softmax layer is used. Then, we can either select the predicted class label as the one receiving the highest probability, o select none if the highest probability is below a certain confidence threshold.

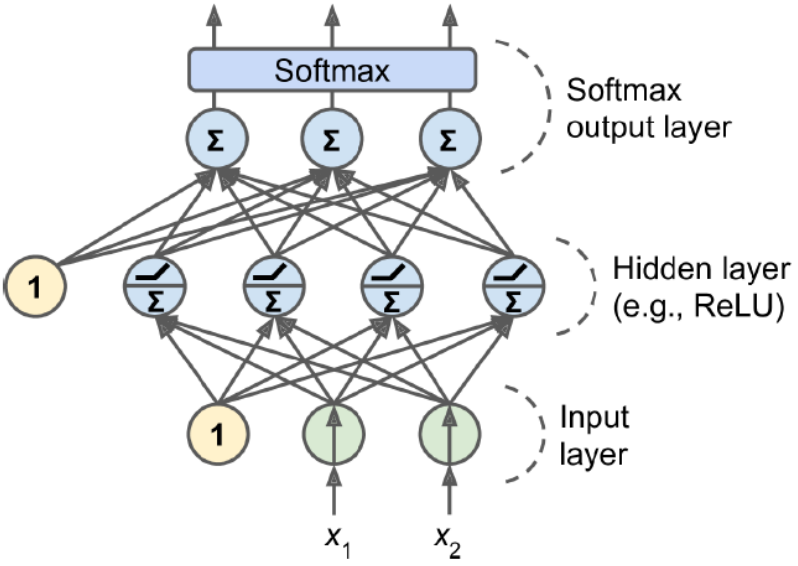

Figure 4.11: A classification neural network. Source: [16]

Convolutional Neural Networks

In order to perform classification tasks on images, all pixels must be fed to the NN as input. This means that the size of the input is equal to the number of pixels that form the image, times three (the number of color channels that characterize each pixel). This makes the process of training NNs for classifying images computationally unpractical.

Therefore, for the task of image classification, *Convolutional Neural Networks* (CNNs) were developed. CNNs attempt to strongly reduce the size of the image without losing the most relevant features, by applying convolution on the input image. More precisely, the convolution layers map the original image to a set of feature spaces which are then used as the input to a neural network (see Figure 4.12).

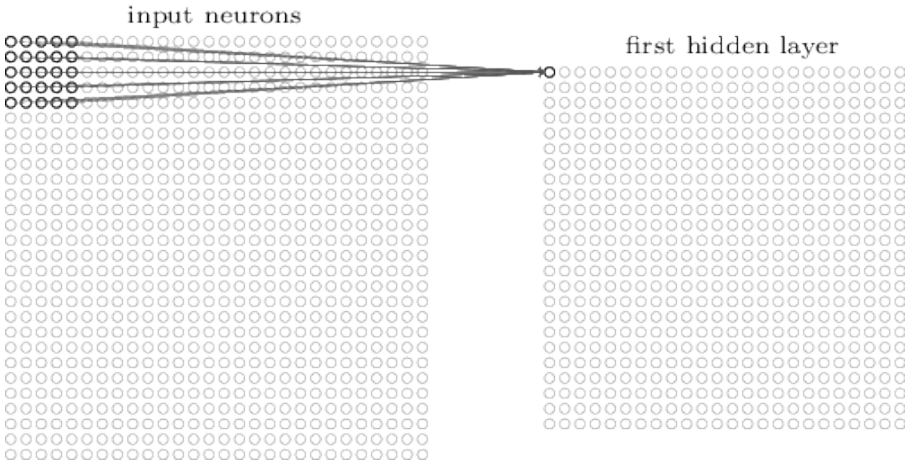

Figure 4.12: Scheme illustrating the convolution on an image. Source: [36]

Real CNNs use a complex combination of several convolution layers as well as other kinds of layers, such as *pooling* layers [36] (see Figure 4.13).

Moreover, several complex architectures were developed starting from CNNs which introduce new layers or connection elements to further improve their training time or the performance.

In particular, the pre-trained NNs used in the applications in this thesis are *Residual* neural networks, also called *ResNet*. The ResNet architecture was first introduced in 2015 and it has

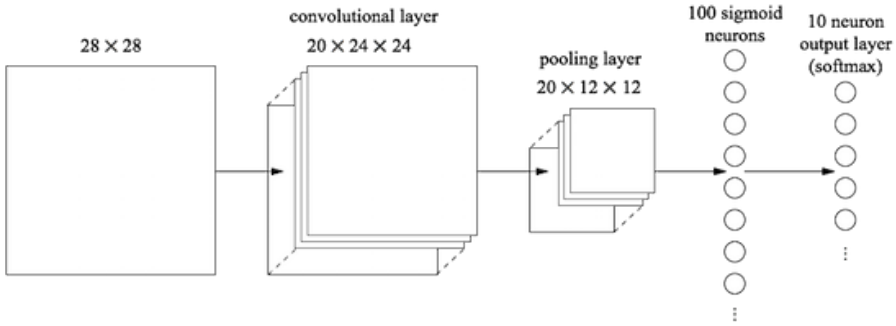

Figure 4.13: Example of a CNN architecture, with several convolution and pooling layers. Source: [36]

since proven excellent ease of training and performance for object classification and detection tasks [22].

4.4.2 Object Detection

The classification tasks that were previously described analyze an image and assign a class label to the entire image (see Figure 4.14). Object detection is a more complex task and it deals with the localization and classification of multiple objects in an image. For example, in the applications presented in this thesis, two object detection tasks must be performed, sometimes simultaneously: face detection and hand detection (identify whether a face or hand is present or not, and localize it/them).

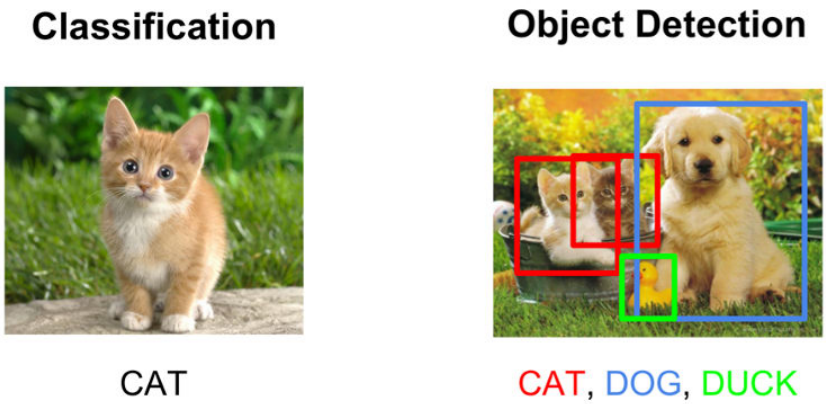

Figure 4.14: Difference between object classification and detection. Source: [25]

In order to achieve object detection, a number of regions of interest (ROIs) must be generated and then analyzed with an image classifier. A basic procedure to generate region proposals is to use a sliding window approach, where a set of boxes with predefined width and height are applied to the whole image repeatedly. This approach has many drawbacks however in that it is computationally expensive and it will detect the same object several times.

Another, more efficient approach was introduced with the Region-based Convolutional Network, or R-CNN [18], which integrates the region proposal phase in the CNN itself (see Figure 4.15). The architecture was then incrementally improved by subsequent work. These

improved architectures were predicatably named by their authors Fast R-CNN [17] and Faster R-CNN [43].

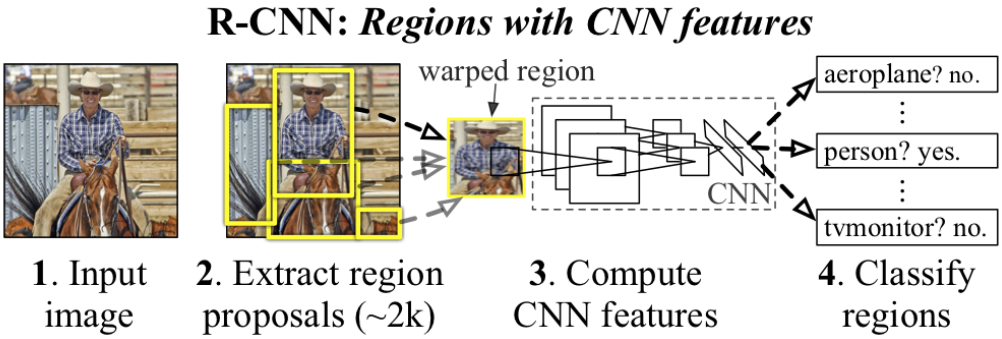

Figure 4.15: High-level scheme of the R-CNN architecture. Source: [18]

The R-CNN architecture can also be combined with other kinds of CNNs to obtain the best detection speed and accuracy. In particular, the ResNet implementation for object detection uses a R-CNN architecture where the original classification CNN is replaced with the ResNet classification network [22]. This is the architecture that was used for the face detection tasks required in the applications presented in 5.

Finally, another family of methods that have demonstrated top-level performance for object detection are *single-shot* networks, such as the *SSD* architecture [33]. In contrast with region-based architectures, single-shot networks eliminate the region-proposal portion relying instead on a set of default boxes. The size variety that objects can have is handled by combining predictions from multiple feature maps, as illustrated in Figure 4.16. This particular architecture is the one that was used for hand detections tasks required in the applications presented in 5.

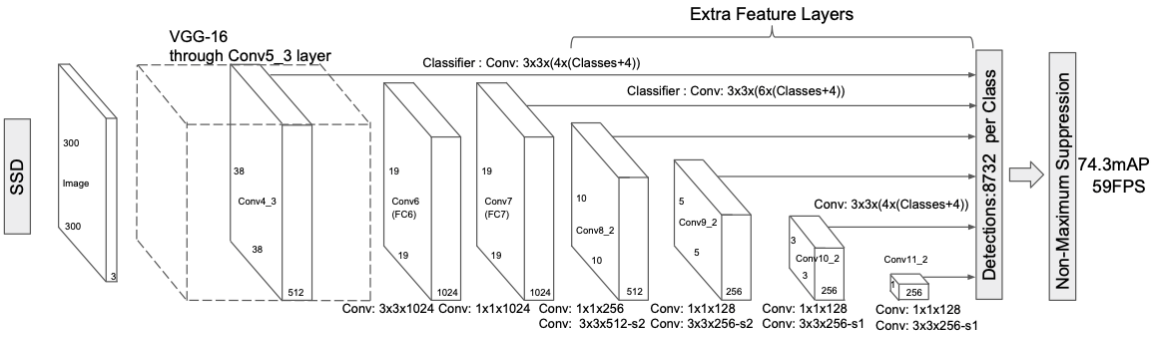

Figure 4.16: High-level scheme of the SSD CNN architecture. Source: [33]

4.4.3 Characterizing shapes with Convex Hulls

The *convex hull* of a shape is the smallest convex set that contains it [46]. The convex hull of the contour of an object is therefore a polygon that contains the whole object, whose vertexes are the convexity points. Between any two consecutive convexity points, a *convexity defect* can be identified, which is the point on the contour being the farthest away from the segment of the convex hull comprised between the two consecutive convexity points [24], [12] (see Figure 4.17).

The convexity points of the convex hull of an object and its convexity defects can be useful in several ways. For example, they were used to identify particular lesions in human organs based on radioisotope images [24]. In this thesis, more simply, they will be used to characterize the shape of a detected hand and identify the correct gesture using simple rules based on their number and on the characteristics.

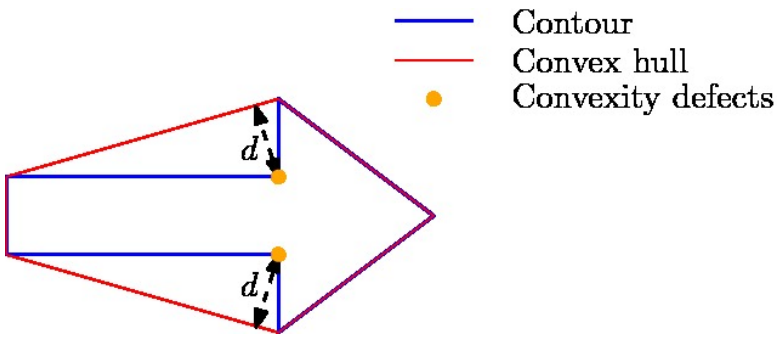

Figure 4.17: Illustration of the convex hull and convexity defects of a contour. Source: [11].

After the hand is identified by a hand detection neural network, the region of interest (a rectangular region containing the hand only) is processed to achieve segmentation (separating the hand from the background). At this point, the contour of the hand can be easily identified. Finally, the convex hull and convexity defects of the hand contour are obtained and gesture recognition is obtained by a set of rules based on them.

Chapter 5

Cases of study

This chapter presents the three cases of study developed for this thesis and illustrates the main methods and algorithms developed for them. The sections are structured as follows: Section 5.1 illustrates the first case of study, Section 5.2 illustrates some methods and algorithms which are common to both the second and third cases of study and Section 5.31 and Section 5.43 illustrate the second and third cases of study, respectively.

5.1 UR arm tracks the face of the user - Television application

The idea beyond this case of study is to create a program usable in a television studio: the television presenter moves in the television studios and the robotic arm, equipped with a camera which focuses on the presenter, follows his movement using a face-tracking algorithm.

The application has been developed in a simulation environment. The robotic arm is simulated in a Polyscope simulation environment, while the video input is acquired by the fixed camera of a laptop. A personal computer (the laptop) processes the video stream and implements the face-tracking algorithm, computing the translation that the robot arm must perform. The personal computer and the simulated robot communicate using the RTDE interface (see Section 2.4.2).

5.1.1 Algorithm overview

The program consists of two sub-programs which run in parallel: a Python code runs on a personal computer and it uses its fixed camera. A Polyscope program, which runs on a virtual machine, simulates the behavior of the robot. In particular, the PC computes the pose to be achieved, while Polyscope gets the computed pose, calculates the joint motion required to achieve that pose and simulates the movement. Figure 5.1 schematizes the PC program, the Polyscope program and their interactions. To obtain the final pose and send it to Polyscope, the PC program implements a face detection algorithm, sets up the RTDE connection used to exchange data and an algorithm used to compute the translation vector which updates the TCP pose, based on the input captured by camera.

5.1.2 Face tracking

In order to track the face position, a face detection algorithm, which is able to recognize a human face in an image frame (see Section 4.4.2), is first needed.

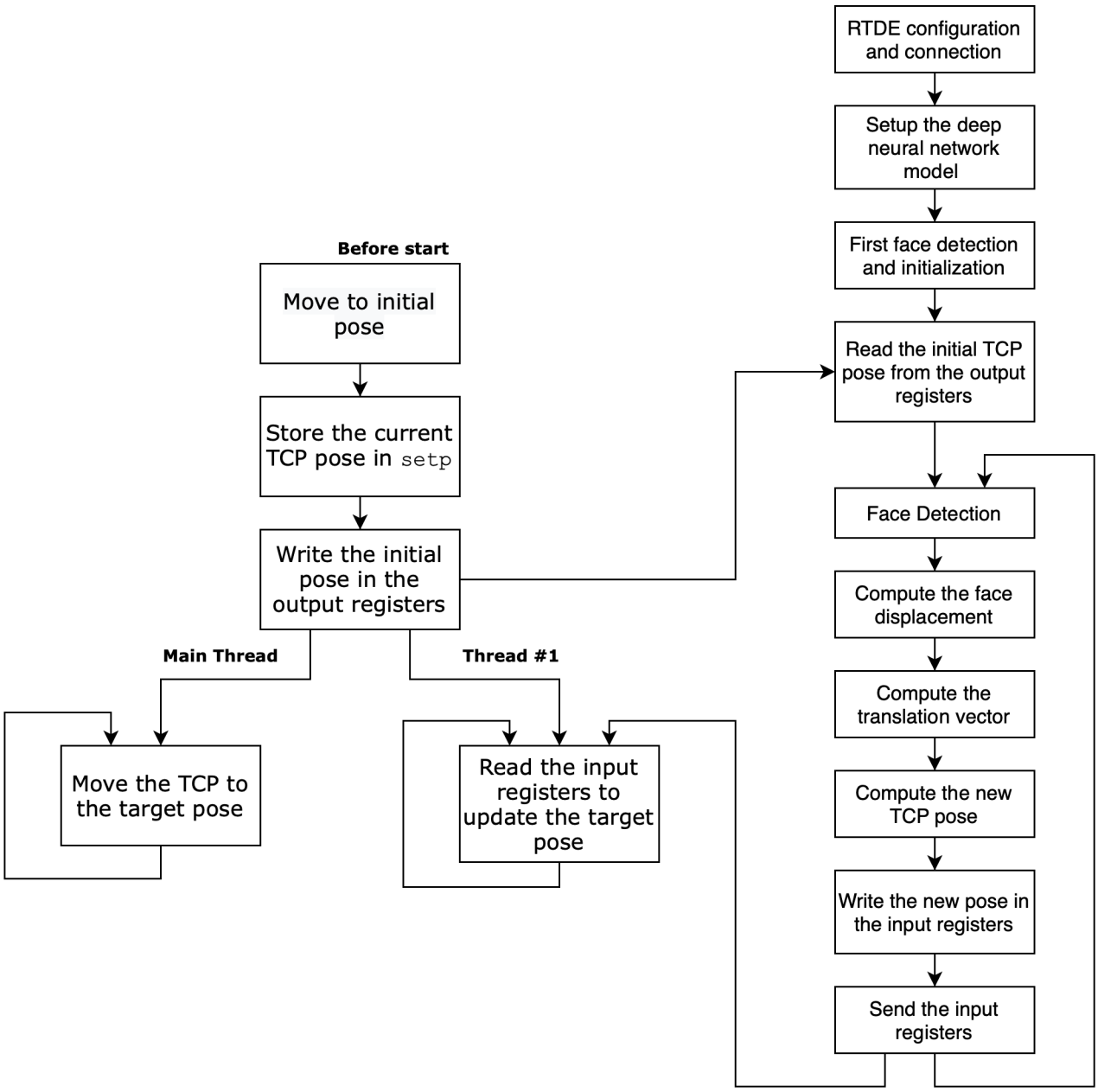

Figure 5.1: Flow chart of the algorithm used to control the robotic arm movement according with the movement performed by an user in the room using a face detection algorithm. On the left, the Polyscope algorithm is shown, on the right, the Python algorithm running on the PC.

To detect the face, a Deep Learning model (DNN) is used. The model is a Region-based Convolutional Neural Network using the ResNet implementation (for more details, see Sections 4.4.1 and 4.4.2). The model is available through the OpenCV GitHub page in several implementations. The implementation used in this thesis is the *Caffe* implementation. Caffe is a deep learning framework developed by Berkeley AI Research and The Berkeley Vision and Learning Center [29].

To correctly use the Deep Learning model and obtain accurate predictions, the input images have to resized to a 300x300 pixels resolutions and the BGR channels must be modified with mean subtraction of values 104 for blue channel, 177 for green channel and 123 for red channel have to be applied [38].

In practice, at first the pre-trained model has to be loaded:

```
net = cv2.dnn.readNetFromCaffe("deploy.prototxt", "
 res10_300x300_ssd_iter_140000_fp16.caffemodel")
```

Listing 5.1: How to load the pre-trained coffee model on Python using OpenCV

The video starts recording the image from the selected source (src=0 refers to the PC's integrated webcam) and the current frame is acquired:

```
vs = VideoStream(src=0).start()
frame = vs.read()
```

Listing 5.2: Read frame from the video

From an image a blob, which is the result of the image preprocessing required to obtain the inputs to the Deep Learning model, can be obtained:

```
blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)), 1.0, (300, 300),
    (104.0, 177.0, 123.0))
```

Listing 5.3: Pre-processing procedure

The image blob is the result of two operations on the original image:

- **Mean subtraction:** as previously mentioned, all pixels of each channel of the image are subtracted by a value which is the mean value for that channel of the training dataset the DNN was trained on;
- **Scaling:** the image is resized to fit the same size the DNN was trained on.

Feeding the blob as input to the DNN, the detections are obtained:

```
net.setInput(blob)
detections = net.forward()
```

Listing 5.4: Detection searching

The output of the DNN is the coordinates of the ROI, which is a rectangle in which the face is contained, and a confidence measure of the detection itself (which quantifies how likely the ROI is to actually contain a face). A rectangle can therefore be drawn around each detected face which has a confidence level higher than a set threshold (a robust and reliable confidence level is 0.55). The confidence of each detected face is specified above the drawn rectangle [53]:

```
if confidence > 0.55:
        box = detection[3:7] * np.array([w, h, w, h])
        (startX, startY, endX, endY) = box.astype("int")
        text = "{:.2f}%".format(detection[2] * 100)
        y = startY - 10 if startY - 10 > 10 else startY +10
        cv2.rectangle(frame, (startX, startY), (endX, endY),(0, 0, 255), 2)
        cv2.putText(frame, text, (startX, y),cv2.FONT_HERSHEY_SIMPLEX, 0.45,
    (0, 0, 255), 2)
```

Listing 5.5: Pre-processing procedure

The figures from 5.2 to 5.7 show the movement of a person in the room and the movement performed by the robot at the same time.

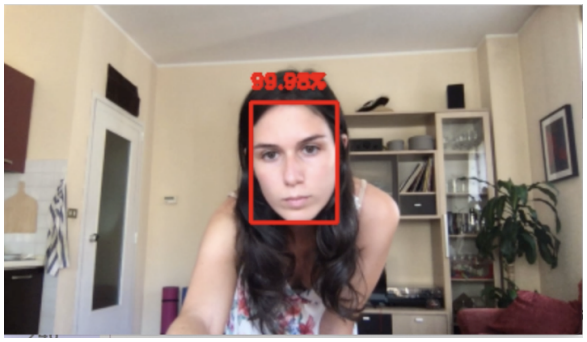

Figure 5.2: Face-detection at the initial time instant.

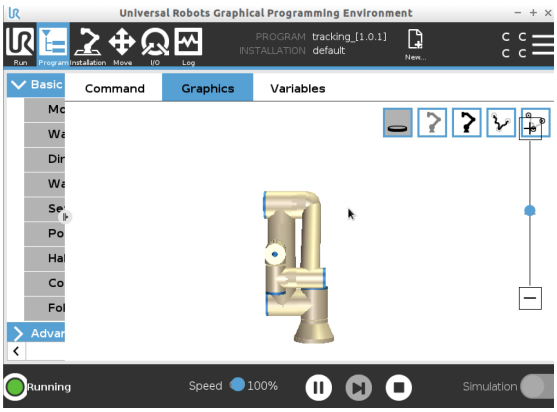

Figure 5.3: Initial robot pose.

Figure 5.4: Face-detection few instants after the first time instant.

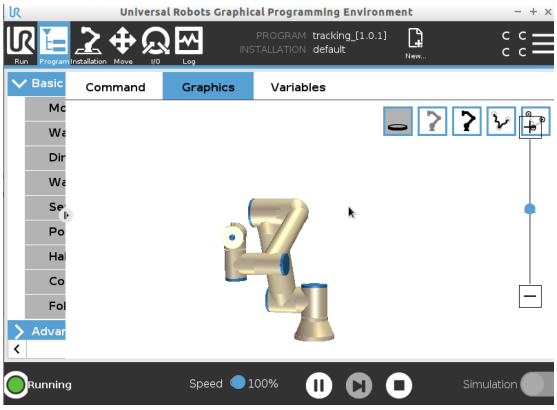

Figure 5.5: Robot pose few instants after the first time instant.

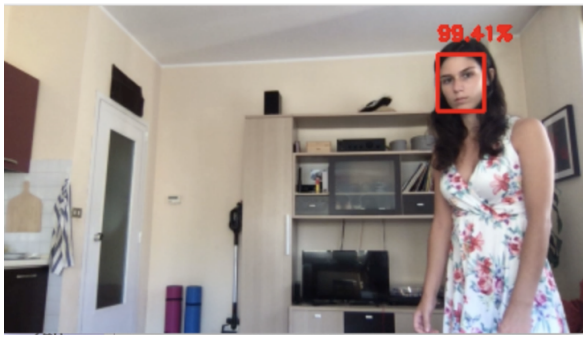

Figure 5.6: Face-detection few instants later.

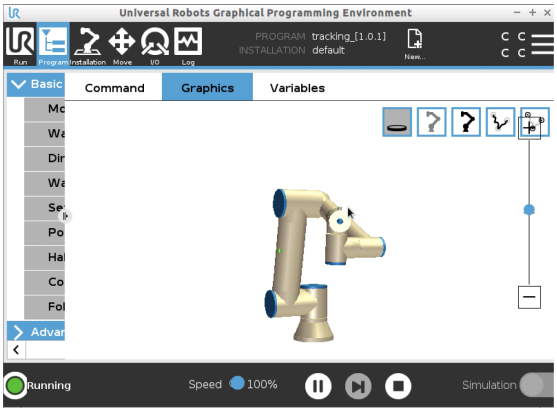

Figure 5.7: Robot pose few instants later.

5.1.3 RTDE and data exchange

As mentioned in Chapter 2.4.2, the Real-Time Data Exchange interface is used to send data between robot and computer. The PC program must set up the connection and define the names of the variables to be exchanged, the robot registers that must be read or written, and the variables data types. This information is encoded in particular structures called *recipes*. The key elements to establish the data exchange are:

- **Robot host:** since the robot is simulated in Polyscope using a virtual machine, the IP address of the robot matches the IP address of the virtual machine;
- **Port number:** the port for RTDE connection is 300004 by default;
- **Configuration file:** the xml configuration file includes 3 recipes about the current state of the robot (state), the pose that the robot has to reach (setp) and the watchdog, as Table 5.1 shows.

Recipes	**Name of the variable to synchronize**	**Types of the variable to synchronize**
State	actual_tcp_pose	VECTOR6D
	output_double_register_0	DOUBLE
	output_double_register_1	DOUBLE
	output_double_register_2	DOUBLE
	output_double_register_3	DOUBLE
	output_double_register_4	DOUBLE
	output_double_register_5	DOUBLE
Setp	input_double_register_0	DOUBLE
	input_double_register_1	DOUBLE
	input_double_register_2	DOUBLE
	input_double_register_3	DOUBLE
	input_double_register_4	DOUBLE
	input_double_register_5	DOUBLE
Watchdog	input_int_register_0	INT32

Table 5.1: RTDE Recipies with names and types of data to synchronize

In particular, the recipe "state" is used to send position data from the robot to the computer, such as the current pose of the TCP which is encoded with 6 float registers that contain the 6 elements defining the pose vector. The recipe "setp" is used to send the pose that the robot has to achieve from the computer to the robot. Once again, the pose is encoded using 6 registers, which represent the 6 coordinates of the robot pose. Finally, the recipe "watchdog" is used to guarantee that the robot action does not miss updates from the PC.

5.1.4 Computation of TCP movements

The initial robot pose, which is the pose with respect to which all the subsequent TCP movements occur, is related to the initial detected face position: in fact, the movements of the TCP along x, y and x axis are computed with respect to the coordinates in which the face appears for the first time in the frame.

In order to avoid that the initial robot position is too close to the positions of maximum extension or bending or near to singularities, if the initial position of the face is too close to the edges of the frame, the initial face position is set to a predefined value, which is different from the real one and which is less close to the corners of the image. In this way, also the initial pose of the robot will be far from any inconvenient pose. This process is performed in Listing 5.6.

```
if 1 < initial_face_center[0] < 100:
    initial_face_center[0] = 100
elif 300 < initial_face_center[0] < 380:
    initial_face_center[0] = 300
else:
    initial_face_center[0] = initial_face_center[0]

if 1 < initial_face_center[1] < 100:
      initial_face_center[1] = 100
elif 160 < initial_face_center[1] < 200:
      initial_face_center[1] = 160
else:
      initial_face_center[1] = initial_face_center[1]
```

Listing 5.6: Algorithm to limit the initial position of the face and thus, the initial position of the robot.

Two different methods are used to compute the displacement along the three axes:

- **Multiplication by a scaling factor:** it is used to compute the displacement along x and y axes;
- **Pinhole camera model algorithm and multiplication by a scaling factor:** they are used to compute the displacement along the z axis.

Lateral movement

The displacement of the TCP of the robot along these axes corresponds to the displacement of the recognized face multiplied by a scaling factor. In particular, using the coordinates of the rectangle which encloses the recognized face and which is shown using the face detection algorithm, the position of the center of the rectangle is computed. The displacement between the current face position and the initial face position is computed by subtracting the coordinate of the center of the rectangle which actually encloses the face and the center of the rectangle which encloses initial face position. The computed displacement along the x and y axes is multiplied by a scaling factor which is experimentally calibrated and its value is 0.002.

Depth movement

To compute the movement along z axis, the pinhole camera model and the multiplication by a scaling factor are used. The pinhole camera model mimics the geometrical projection of a pinhole camera. A pinhole camera is a basic camera which consists of a closed box which has a single tiny hole on a wall, which allows light to enter. Light hits a photosensitive surface inside the box and produces an upside-down projection of what is outside of the box on its back wall, as Figure 5.8 illustrates.

In the pinhole camera model, the aperture of a camera is reduced to a single point which is called center of projection and the photosensitive surface, assumed to be planar, is called the image plane. To determine where a 3D point is depicted in the image, a straight line which intersects this point optical center is drawn and the image of the 3D point is obtained at the intersection of the previously drawn line with the image plane, as Figure 5.9 shows. As previously mentioned, the image plane shows the upside-down projection of objects. To avoid

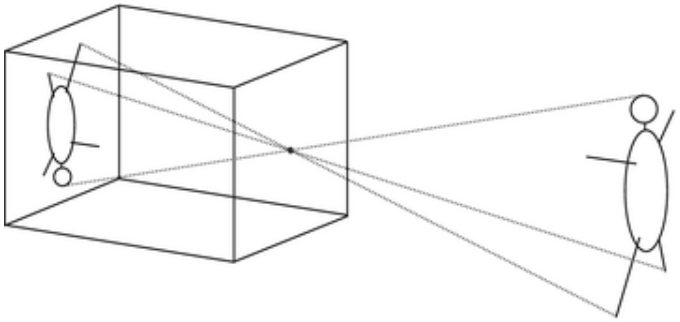

Figure 5.8: Sketch of the pinhole camera. Source: [26]

this inversion, the true image plane can be replaced with a virtual plane placed at the same distance from the center as the original image plane and parallel to it. To understand the

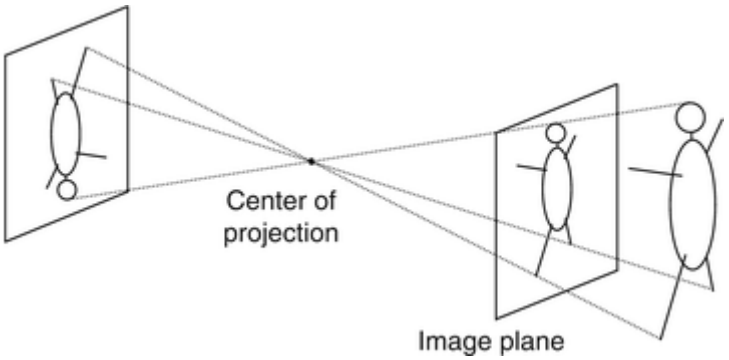

Figure 5.9: Sketch of the pinhole camera model. Source: [26]

algebraic formulation of the pinhole camera model, some concepts must be introduced first:

- **Focal length (f):** the distance between the center of projection and the image plane;
- **Optical axis:** the line which passes through the optical center and which is orthogonal to the image plane;
- **Principal point:** the point given by the intersection between the optical axis and the image plane;
- **3D point coordinates (X, Y, Z):** they are expressed in the camera coordinate system, whose origin is in the optical center; Conventionally, the Z axis is coincident with the camera's optical axis, and the X and Y axes are parallel to the columns and rows of pixels in the image plane;
- **Image coordinates system (x, y):** its origin is in the principal point. The x and y axes are parallel to the X and Y axes respectively;
- **Pixel coordinate system (u, v):** its origin lies in one of the image area's corners. According with the convention adopted by OpenCV it matches the upper left corner of the image. The u and v directions are parallel to x and y axes respectively; their value is expressed by the number of pixels.

Figure 5.10 shows more in detail the pinhole camera models, based on the concepts previously expressed [26].

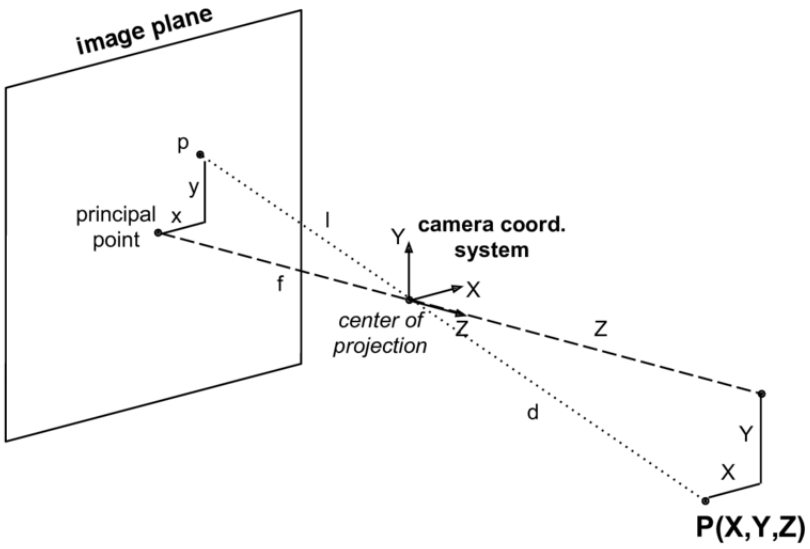

Figure 5.10: Pinhole camera model with a point P(X,Y,Z) according to the camera coordinate system. Source: [52]

Comparing similar triangles, the image point's coordinates expressed in the image coordinate system are:

$$x = f \cdot \frac{X}{Z} \tag{5.1}$$

$$y = f \cdot \frac{Y}{Z} \tag{5.2}$$

Thus, the focal length and the real point coordinate along the Z axis can be computed respectively as:

$$f = Z \cdot \frac{x}{X} \tag{5.3}$$

$$Z = f \cdot \frac{X}{x} \tag{5.4}$$

To experimentally determine the focal length, it is sufficient to measure x, which is the size of the face projected in the image plane along the x axis, expressed in pixels, Z which is the distance between the origin of the camera reference system and the origin of the coordinate system of the face in the real world and X, which is the real dimension (in meters) along the X axis of the face. The experimentally evaluated values are:

$$x = 88\,pixels \quad Z = 0.495\,m \quad X = 0.16\,m \tag{5.5}$$

Having such data, the focal length of the used camera was computed as:

$$f = Z \cdot \frac{x}{X} = 272.5 \tag{5.6}$$

Finally, the point position along the Z axis will depend on the value of the point along the x axis and it can be computed as:

$$Z = f \cdot \frac{X}{x} = 272.25 \cdot \frac{0.16}{x} \tag{5.7}$$

In order to limit the movement along the Z axis of the robotic arm (in such a way that when the user moves for example 2 meters or more along the Z axis, the robotic arm follows

the user without reaching its full extension), the obtained value is multiplied by a scaling factor which was set to 0.2. In a real world application, this parameter would be set according to the operational requirements of the studio.

Translation vector computation

To compute the translation along x, y and z axes, two functions are designed as part of the PC program. The function `translation_vector` computes the translation vector without applying any scaling factor. The vector `translation` has three elements: `translation[0]` corresponds to the translation along x axis, `translation[1]` along y axis and `translation[2]` along z axis.

```
def translation_vector(face_center, initial_face_center, translation_bool,
    y_rect, initial_y_rect):
    face_center_opp = [face_center[0], -face_center[1]]
    initial_face_center = [initial_face_center[0], -initial_face_center[1]]

    translation = [0,0,0]
    translation[0] = face_center_opp[0]-initial_face_center[0]
    translation[1] = face_center_opp[1]-initial_face_center[1]
    if y_rect>0 and initial_y_rect:
        translation[2] = (real_face_y*focal_length)/y_rect-(real_face_y*
    focal_length)/initial_y_rect
    else:
        translation[2] = 0
    translation = translation * translation_bool

    return translation
```

Listing 5.7: Function to compute the translation vector without any scaling factor

In particular:

- `translation_bool` is a boolean variable which is set to 1 when a face is detected and to 0 when there is no face in the frame;
- `y_rect` is the dimension of the rectangle which encloses the face along y axis in pixels;
- `initial_y_rect` is the `y_rect` computed the first time in which a face appears in the video;

The sign inversion along the x and y axis computed in `face_center_opp` is due to the fact that the webcam of the computer flips the image (flipping the right and left side) and to the orientation of the y axis according to the OpenCV convention (the origin of the axis is in the upper left corner and the y axis is oriented downwards).

The function `new_robot_pose` computes the next pose of the TCP of the robot adding the scaled translation vector to the initial TCP pose.

```
def new_robot_pose(translation, initial_robot_position):
    translation_list = list(translation)

    scaled_translation_vector = [0,0,0]
    scaled_translation_vector[0] = scaling_factor*translation_list[0]
    scaled_translation_vector[1] = scaling_factor*translation_list[1]
    scaled_translation_vector[2] = scaling_factor_y*translation_list[2]

    new_robot_position[0] = initial_robot_position[0]+
   scaled_translation_vector[0]
```

```
10      new_robot_position[1] = initial_robot_position[1]-
        scaled_translation_vector[2]
11      new_robot_position[2] = initial_robot_position[2]+
        scaled_translation_vector[1]
12
13      return new_robot_position
```

Listing 5.8: Function to compute the next TCP pose of the robotic arm

5.1.5 Robot program

The robot program, which runs in the Polyscope simulation environment, is responsible of reaching the desired TCP pose and the algorithm to accomplish this task is shown in Figure 5.11.

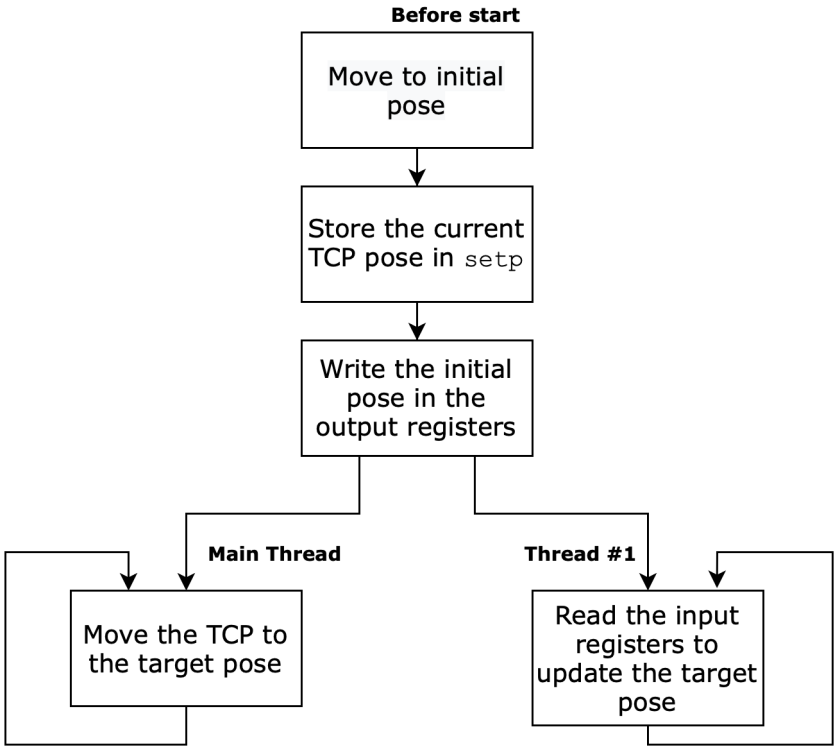

Figure 5.11: Flow chart of the algorithm which Polyscope runs

Analyzing more in depth the code presented in Listing 5.9, in the `Before Start` sequence, as the flow chart shows, elementary operations are performed, such as moving to the initial pose and its storage in the variable `setp`, setting of the watchdog and writing of the `setp` variable in the output float register: in this way the PC program can read the initial pose from these registers. In particular, line `5` activates the watchdog for the RTDE input variable `input_int_register_0`. If the watchdog does not receive an input update for this variable in the time period specified by the second argument of the function, the action specified as the third argument of the function is performed. In this case, if the update frequency is less than $1\,Hz$, the program will stop.

In the `Robot Program`, which is the main thread of the algorithm, the new pose is reached through two functions [3]:

- `get_inverse_kin(tool pose)`: it calculates the inverse kinematic transformation, from tool space to joint space. It returns the solution closest to the current joint positions.

- `servoj(output of get_inverse_kin, lookahead time, gain)`: it is used for online realtime control of point positions. The lookahead time is a parameter used to project the current position forward in time maintaining the current velocity. The gain adjusts the current position towards the desired one.

Meanwhile, `Thread 1` stores the current TCP pose in the `setp` variable and reads the next pose to reach, which is computed by the PC program, from the input float registers. The pose to reach is stored in the `setp` variable. It is then used by the `Main Program` and the new pose is achieved by the robotic arm.

```
BeforeStart
  MoveJ
    init_pos
  setp:=get_actual_tcp_pose()
  rtde_set_watchdog("input_int_register_0",1,"PAUSE")
  Loop 1 times
    init:=init_pos
    write_output_float_register(0, init[0])
    write_output_float_register(1, init[1])
    write_output_float_register(2, init[2])
    write_output_float_register(3, init[3])
    write_output_float_register(4, init[4])
    write_output_float_register(5, init[5])
    sync()
Robot Program
  setp_inv_kin:=get_inverse_kin(setp)
  servoj(setp_inv_kin, lookahead_time = 0.05, gain = 500)
Thread_1
  setp:=get_actual_tcp_pose()
  Loop
    tmp:=p[0,0,0,0,0,0]
    tmp[0] = read_input_float_register(0)
    tmp[1] = read_input_float_register(1)
    tmp[2] = read_input_float_register(2)
    tmp[3] = read_input_float_register(3)
    tmp[4] = read_input_float_register(4)
    tmp[5] = read_input_float_register(5)
    If tmp != p[0,0,0,0,0,0]
      setp:=tmp
    sync()
```

Listing 5.9: Function to compute the next TCP pose of the robotic arm

5.2 Gesture recognition algorithm

This section describes the gesture recognition algorithm which is common to the second (Section 5.31) and third (Section 5.43) case of study. The second case of study consists in an industry application: the program allows an operator to select the object to be processed by the robot arm by performing a simple hand gesture in front of a camera. The third one is a health-care application: a patient which cannot move due to health problems, performs gestures which represent the object which he/she needs and the robot will hand it to him.

5.2.1 Algorithm overview

Similarly to the first case of study, the webcam of the PC is used, and once again, in order to extract the region of interest (or ROI) from the entire image captured by the camera a

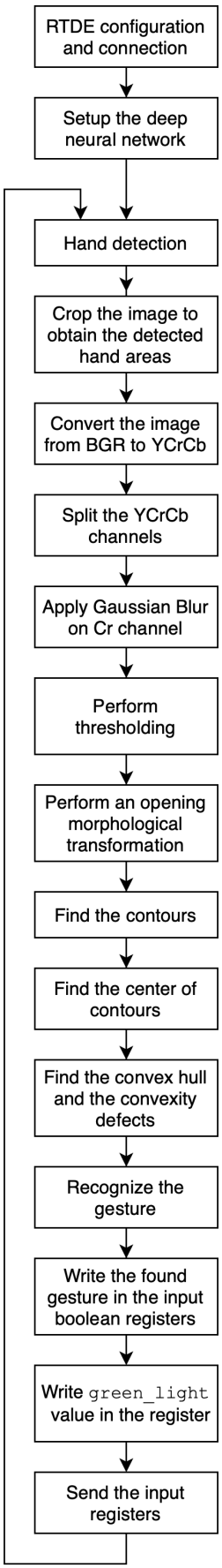

Figure 5.12: Flow chart of the PC algorithm for gesture recognition

pre-trained Deep Learning model is used. This time, however, the ROI is defined by the region around the hand of the operator/patient, and therefore a DNN trained for hand detection must be used.

Once the hand is detected and the ROI extracted, several Computer Vision techniques are used to perform proper gesture recognition (i.e. identify the gesture which a person made).

Figure 5.12 shows the flow chart of the operations which have to be performed to recognize gestures. Once the gesture to perform is recognized, it is sent to the robot program (running in the Polyscope).

Since the Polyscope interface cannot simulate a realistic environment with people and objects to process, the RoboDK simulator is used. In particular, the movement of the robot are performed on Polyscope and, at the same time, are emulated in RoboDK.

The setup for the second and third cases of study is therefore as follows: a PC (a laptop) acquires video through its integrated camera and processes it. The computer program performs hand detection and gesture recognition, and sends the gesture to be performed to the simulated robot through the RTDE interface. The Polyscope simulation environment runs the robot program and simulates the robot movement. The RoboDK simulation environment mirrors the movements performed by the robot in Polyscope and reproduces them in a more complex and realistic simulation environment.

5.2.2 Hand detection algorithm

The hand detection algorithm used in this work is developed by the researcher Victor Dibia [13] using the *egohands* dataset from Indiana University [4] which contains 48 videos about first-person interactions between two people.

The model is trained using *Tensorflow*, in particular the object detection API. In order to use it, the training data are converted in Tensorflow format. The dataset is spilt into train folders (80% of data), test (10% of data) and evaluation folder (10% of data). A detailed overview of the algorithm is described in Section 4.4.2.

In Figure 5.13 an example frame captured by the webcam in which both the face and a hand are recognized is shown.

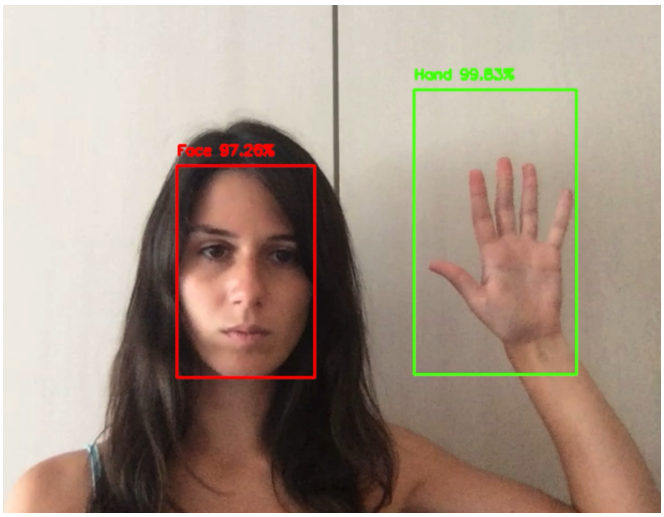

Figure 5.13: Hand and face recognition using the webcam frame

5.2.3 Gesture recognition

Thanks to the hand detection algorithm output, the original image frame can be cropped in order to obtain an image which shows only the ROI in which the detected hand is present. An example is shown in Figure 5.14).

Figure 5.14: Original frame region of interest in BGR color space

After the ROI is extracted, gesture recognition is performed based on some geometrical features which encode its shape. In particular, the gesture is identified base on the characteristics of the convex hull and the convexity defects (see Section 4.4.3) of the hand's contour. However, in order to extract the hand's contour, its image must first be processed. First, the image is converted to the YCrCb color space and the red-difference chroma channel Cr only is retained. Then, the image is denoised using a Gaussian blur filter. Next, a binary image which effectively separates the hand from the background is obtained with a thresholding operation and a morphology transformation is applied to smooth out the obtained shape. Finally, the contours can be easily extracted from the binary image of the hand.

As explained in Section 5.15, converting the BGR image in YCrCb means to separate the luminance (Y channel) from the chromatic aspect (Cr and Cb channels) of the color of the image (see Figures from 5.16 to 5.18). Considering only the chromatic aspect allows to perform easier skin recognition, as image 5.15 shows.

Figure 5.15: YCrCb color space transformation of the original frame

The code in Listing 5.10 shows how the YCrCb transformation of the original image and channel split was performed using the OpenCV library (in Python).

```
1 img_YCrCb = cv2.cvtColor(crop_img, cv2.COLOR_BGR2YCrCb)
2 (ch_y, ch_cr, ch_cb) = cv2.split(img_YCrCb)
```

Listing 5.10: Color space transformation from BGR to YCrCb and channel split

Figure 5.16: Y channel of YCrCb color space.

Figure 5.17: Cr channel of YCrCb color space.

Figure 5.18: Cb channel of YCrCb color space.

In this work, only the red-difference chroma channel Cr of the two chromatic aspects is considered. In fact, considering only this channel is sufficient to perform a correct skin detection [1], as Figure 5.17 shows.

To suppress noise from the Cr channel, a Gaussian Blur is applied. An overview of this filter can be found in Section 4.3.2. Using this filter, the intensity of each pixel is substituted by a weighted sum of the intensities of pixels of the original image in the surrounding area, as Figure 5.19 shows. The function to perform this operation is shown in Listing 5.11.

```
1 ch_cr = cv2.GaussianBlur(ch_cr, (5, 5), 1)
```

Listing 5.11: Gaussian blur

where the first argument of the function is the source, the second is referred to the kernel dimension and the third element is the standard deviation along x axis.

Figure 5.19: Gaussian blur filter applied to the Cr channel of the YCrCb frame.

To separate the hand from the background a thresholding operation is performed. The result is shown in Figure 5.20.

Figure 5.20: Thresholding technique to separate the hand from the background.

Listing 5.12 allows to obtain a good result performing binary thresholding with automatic threshold selection using Otsu's algorithm.

```
1 _, ch_cr_bin = cv2.threshold(ch_cr, 0, 255, cv2.THRESH_BINARY + cv2.
    THRESH_OTSU)
```

Listing 5.12: Thresholding

Here, the first argument of the function is the source image, the second argument indicates that the threshold value is not specified since it is defined by Otsu's algorithm (which selects the best threshold value), the third is the value that will be assigned to the output pixels which are greater than the threshold value, and the last element is referred to the name of the chosen algorithm.

If the obtained shape is still noisy, the morphological transformations may allow to attenuate the noise on the image and thus, to obtain simpler and cleaner topology of the image as the Section 4.3.3 explains. In the Figure 5.21 slight differences with respect to Figure 5.20 can be noticed. More noisy shapes would benefit more from this morphological transformation.

Figure 5.21: Opening morphological transformations.

In Listing 5.13, the morphological transformation performed with OpenCV is shown.

```
1 kernel = np.ones((10, 10), np.uint8)
```

```
image_bin = cv2.morphologyEx(image_bin, cv2.MORPH_OPEN, kernel)
```

Listing 5.13: Morphological transformation

Here, the first argument of the function is the source, the second is referred to the type of morphological operation to apply and the last is the kernel which is the structuring element used for dilation.

To find the contours of the binary image, the Listing 5.14 can be used.

```
cont_all, hier = cv2.findContours(image_bin, cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)
```

Listing 5.14: Contour finding

The first argument is the source, the second is the contour retrieval mode and the third is the contour approximation method. Using `RETR_EXTERNAL` retrieves only the extreme outer contours. Using `cv2.CHAIN_APPROX_SIMPLE` the algorithm compresses horizontal, vertical and diagonal segments and leaves only their end points.

The quality of the found contours depends on the threshold image obtained in the previous step. To exclude the contours of small area which can be present inside or around the picture, Listing 5.22 is implemented.

```
def contour_filtering(cont_all, area_to_ignore):
    c_bigger = []
    for i in range(len(cont_all)):
        cont = cont_all[i]
        a = cv2.contourArea(cont)
        if a > area_to_ignore:
            c_bigger.append(cont)
    return c_bigger
```

Listing 5.15: Contour filtering function

If the area of the considered contour is bigger than `area_to_ignore`, the contour is accepted. The `area_to_ignore` is set to 1/3 of the image area as Listing 5.16 shows.

```
cont_all_big = contour_filtering(cont_all, (rows*cols)/3)
```

Listing 5.16: Contour filtering

To draw the contour on the binary image Listing 5.17 is used. It allows to visualize on the binary image `image_extract_3ch` the contours `cont_all_big`.

```
image_extract_3ch = np.zeros((rows, cols, 3), dtype='uint8')
cv2.drawContours(image_extract_3ch, cont_all_big, -1, (0, 0, 125), -1)
```

Listing 5.17: Contour filtering

The first argument is the image on which the contour is drawn, the second is the contour, the third refers to the contour index (-1 value means that the all contours are drawn), the fourth refers to the contour color and the last to the thickness of the contour (-1 value means to fill the area inside the contour). The obtained output is shown in the Figure 5.22.

Figure 5.22: Contour drawing.

For the purpose of gesture recognition, the computation of the center of the contours have a certain relevance. To compute its position, at first the moments have to be estimated. The moment, computed using Green's theorem, describes some statistical properties of the shape, such as the area, the centroid and the orientation. As shown in Listing 5.18, the center of contours can be estimated from the moments.

```
for c in cont_all_big:
  M = cv2.moments(c)
  cX = int(M["m10"] / M["m00"])
  cY = int(M["m01"] / M["m00"])
  cv2.circle(image_extract_3ch, (cX, cY), 15, (255, 255, 255), -1)
```

Listing 5.18: Contours center estimation

Drawing a white circle in the contours center of the previous image, the obtained output is shown in Figure 5.23.

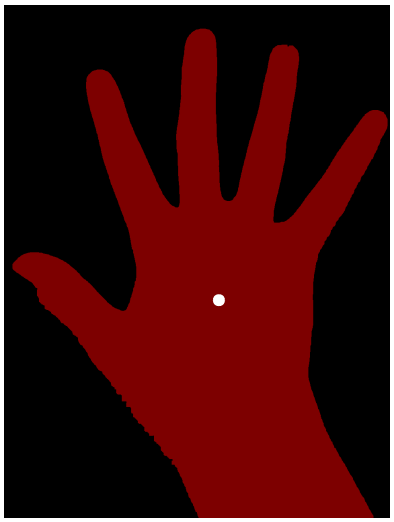

Figure 5.23: Contours center drawing.

To identify gestures, the convex hull (see Section 4.4.3) and the convexity defects are extracted from the hand contour, as shown in Listing 5.19.

The key of the gesture recognition algorithm is that convexity points (the vertices of the convex hull) are assumed to coincide with the tips of the fingers, while the convexity defects

are assumed to be present at the concavity where the fingers meet the palm. Thus, thanks to the convex hull it is possible to figure out how many fingers are extended and their relative angle, which also allows to guess exactly which fingers are extended.

```
cnt = cont_all_big[0]
hull = cv2.convexHull(cnt, returnPoints=False)
defects = cv2.convexityDefects(cnt, hull)
if defects is not None:
  for i in range(defects.shape[0]):
          s, e, f, d = defects[i, 0]
          d /= 256
          if d > rows * far_ratio:
               start = cnt[s][0]
               end = cnt[e][0]
               far = cnt[f][0]
               cv2.line(image_extract_3ch, tuple(start), tuple(end), [0, 255,
    0], 2)
               cv2.circle(image_extract_3ch, tuple(far), 5, [255, 0, 0], -1)
               contour_with_defects.append([start, far, end])
               contours_with_defects.append(contour_with_defects)
```

Listing 5.19: Convex hull and convexity defects estimation

If the algorithm finds more than one `cont_all_big` (which might happen in harsh light condition or when the background has a color similar to the hand), it selects only the first detected contour which should be the most reliable.

In the `cv2.convexHull()` function, the first element is the contour and the second is a flag: if it is `True`, it returns the convex hull points. To find the convexity defects, this flag has to be set to `False`.

In function `cv2.convexityDefects()`, the first argument is the contour and the second the convex hull. `defects` returns an array where each row contains values a start point (`start`), an end point (`end`), a farthest point (`far`) and an approximate distance to farthest point (`d`). To clarify, these points are represented in Figure 5.24 and 5.25.

The convexity points and defects are what allows to establish how many and which fingers are extended. However, due to the rounded (and possibly, still noisy) nature of the hand's shape, many unrelevant convexity points and defects may be found, and they must be filtered out in order to ensure the robustness of the algorithm. To exclude these unrelevant defects from the obtained array, Listing 5.19 applies a filter which is used to make sure that the distance between a convexity defect and the next is bigger than the minimum distance which the physiognomy of the hand allows to have. This parameter is chosen experimentally.

Iterating over all the defects, the complete drawing of the convex hull (green line) and the convexity defects (blue points) are obtained, as the Figure 5.26 shows.

Sometimes the algorithm might detect more than one convex hull points on a single finger (for example, in certain conditions the computation of the start point of a finger and the end point of the previous finger might not coincide) and this could compromise the effectiveness of this algorithm. In order to avoid it, an additional filter is introduced: `filter_vertices_by_angle` [40]. It uses the law of cosines to compute the angle between 3 points. The law of cosines states that *"the square of the length of any side of a triangle equals the sum of the squares of the length of the other sides minus twice their product multiplied by the cosine of their included angle"*. Thus, considering the Figure 5.27, to compute the α angle:

$$a^2 = b^2 + c^2 - 2bc\ cos(\alpha)) \tag{5.8}$$

$$\alpha = cos^{-1}\left(\frac{b^2 + c^2 - a^2}{2bc}\right) \tag{5.9}$$

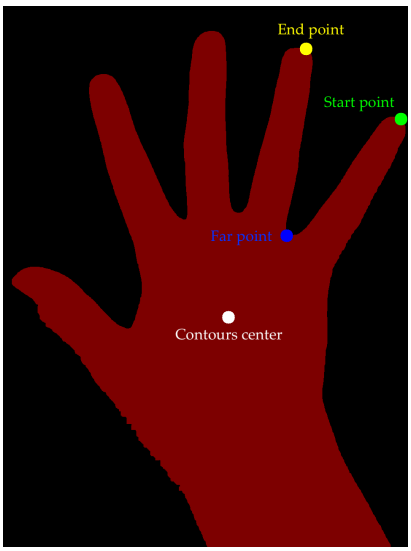

Figure 5.24: Representation of start, farthest and end points.

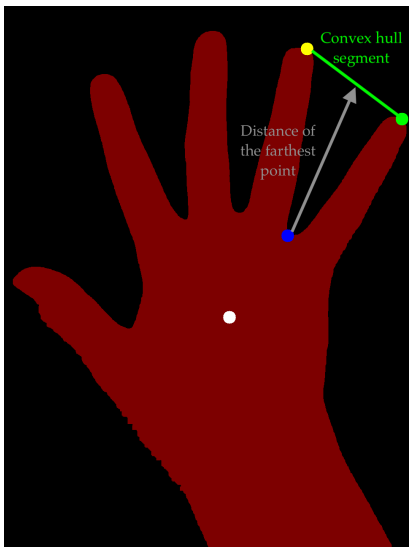

Figure 5.25: Representation of the distance to farthest point with respect to the convex hull.

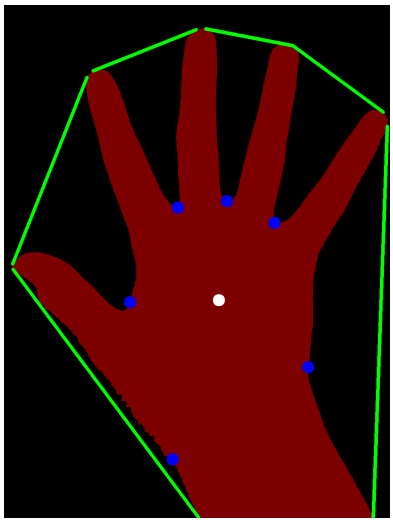

Figure 5.26: Convex hull and convexity defects drawing.

A new list, which contains only the `contour_with_defects` elements which satisfy the condition set by this filter, i.e. the angle between two consecutive points is less than a certain maximum value, is obtained[40] (see Listing 5.20). In particular, `max_angle` is selected by trial-and-error and it is set to 80°.

```
def filter_vertices_by_angle(triple, max_angle):
  a = linalg.norm(triple[0] - triple[2])
  b = linalg.norm(triple[1] - triple[2])
  c = linalg.norm(triple[1] - triple[0])
  angle = np.arccos(((b ** 2 + c ** 2 - a ** 2) /(2 * b * c))) * (180 / np.
   pi)
  return angle < max_angle
```

Listing 5.20: Angles filtering function

The first argument of the function, `triple`, is composed from the farthest point of a finger i, the end and far points of the previous finger i-1 (see Listing 5.21). In Figure 5.28 an example of the points which compose a triple is shown.

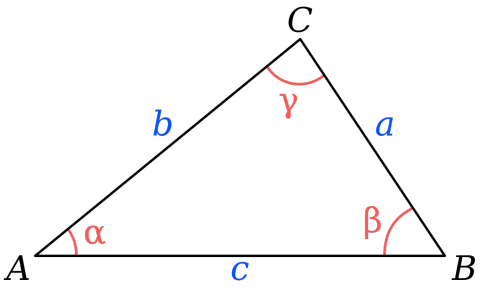

Figure 5.27: Law of cosines representation. Source:[55]

```
for i in range(len(contour_with_defects)):
  triple1 = contour_with_defects[i]
          triple2 = contour_with_defects[i-1]
          new_triple = [triple1[1], triple2[2], triple2[1]]
```

Listing 5.21: Creation of the list called triple

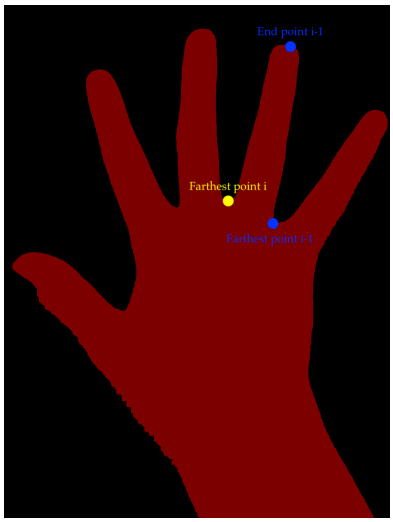

Figure 5.28: Representation of the points of a triple.

To ensure that two consecutive triples are connected, i.e the start point of the current triple coincides with the end point of the previous triple and the distance between two consecutive farther points is limited, Listing 5.22 is also used [40].

```
def check_mask_cutoff(triple1, triple2):
  return (triple1[0][0] == triple2[2][0] and abs(triple1[0][1] - triple2
    [2][1]) > 60 or triple1[0][1] == triple2[2][1] and abs(triple1[0][0] -
    triple2[2][0]) > 60)
```

Listing 5.22: Consistence of triples checking

Now, applying the previous filters and a final filter which limits the maximum number of detected fingers, it is possible to obtain a robust computation of the number of finger, as Listing 5.23 shows.

```
if n_finger<=4 and filter_vertices_by_angle(new_triple, 80):
  n_finger += 1
  top_finger_list.append(new_triple[1])
```

Listing 5.23: Number of fingers computation

The list `top_finger_list` then contains the coordinate of the tip of each detected finger.

To define the gesture type, a defining parameter is the angle between two consecutive fingers. Having the coordinate of the center and the coordinate of the top of each finger, the distance between these two coordinates can be easily computed. Having these distances, the angle can be computed using the definition of the scalar product between two vectors, whose definition is given by equation 5.10.

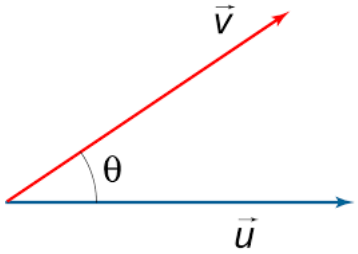

Figure 5.29: Representation of the angle between two vectors.

$$\vec{u} \cdot \vec{v} = |\vec{u}| \cdot |\vec{v}| \; cos(\theta) \tag{5.10}$$

$$cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \cdot |\vec{v}|} \tag{5.11}$$

This computation is performed in the PC program as in Listing 5.24.

```
for i in range(1, len(top_finger_list)):
  dist_center_finger1.append(top_finger_list[i]-(cX,cY))
  dist_center_finger2.append(top_finger_list[i-1]-(cX,cY))
  cosine_angle = (np.dot(dist_center_finger1[i], dist_center_finger2[i])/(np
   .linalg.norm(dist_center_finger1[i])*np.linalg.norm(dist_center_finger2[i
   ])))
  angle.append((math.degrees(np.arccos(cosine_angle))))
  angle_diff.append(angle[i]-angle[i-1])
```

Listing 5.24: Angle between finger computation

To make sure that the computed angles are correct, another check is performed: an angle is accepted only if it is less than 90°, since the human hand cannot produce angles greater than 90° under any circumstances. This operation is performed in Listing 5.26.

```
n_angle = 0
for j in range(len(angle_diff)):
if angle_diff[j] < 90:
  cv2.circle(image_extract_3ch,tuple(top_finger_list[j]), 5, [0, 255, 255],
   -1)
  line = cv2.line(image_extract_3ch, (cX, cY), tuple(top_finger_list[j]),
   [0,255,255], 2)
  n_angle +=1
```

Listing 5.25: Filtering of the previously obtained angles

If an angle which satisfies all the requirement is found, for illustration purposes, a circle can be drawn at the tip on the finger and a line can be drawn between this point and the center of the contours, as Figure 5.30 shows.

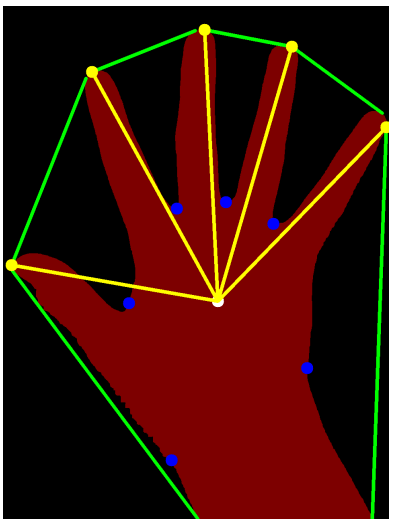

Figure 5.30: Representation hand with the detected fingers.

5.3 Gesture recognition in an industrial context

Sometimes the workpiece on which an operator has to process might be distant from the position of the operator or the workpiece might be heavy or sharp and thus potentially dangerous for the health of the operator. In these cases, a collaborative robot could help in making the work environment safer and it could facilitate the operator's work.

In this workplace, the five workpieces are far from the operator which has to executes some operations on them. After the processing is completed, the workpieces have to be moved in another station.

The workpieces are numbered in increasing order from the right to the left. According to the gesture performs by the operator, the correct piece is taken.

Figure 5.31 shows a complete overview of the programs which run in parallel on the PC, Polyscope and RoboDK and which will be illustrated in the following sections.

5.3.1 RTDE and data exchange

In Section 2.4.2 the actions required to set up the RTDE connection are explained. The only element which changes with respect to the previous case (Section 5.1) of study is the composition of the recipes defined in the configuration file. The `output_bit_register_72` is a boolean output register which is set to `True` if the gripper has to attach a certain workpiece, otherwise it is set to `False`. The `output_bit_register_73` controls the detaching of an object: it is set to `True` if the gripper has to detach a workpiece, otherwise it is set to `False`. Since also RoboDK uses the RTDE interface to read the data elaborated by the simulated robot in Polyscope, these two registers are used to manage the attaching and the detaching action in this simulator. The `input_bit_register_65` indicates that the gesture is 'None', the registers from `input_ bit_register_66` to `input_bit_register_70` indicates the gestures from 'One' to 'Five' in increasing order and the register `input_bit_register_71` is a binary register which indicates if the information about the gesture to perform is sent or not to the robot. The role of the watchdog recipe is the same explained in the Section 5.1.3.

5.3.2 Five number gesture recognition

Starting from the code explained in the Section 5.2.3 and analyzing the number of detected fingers and the angle between the segments which connects the contour center to the tip of

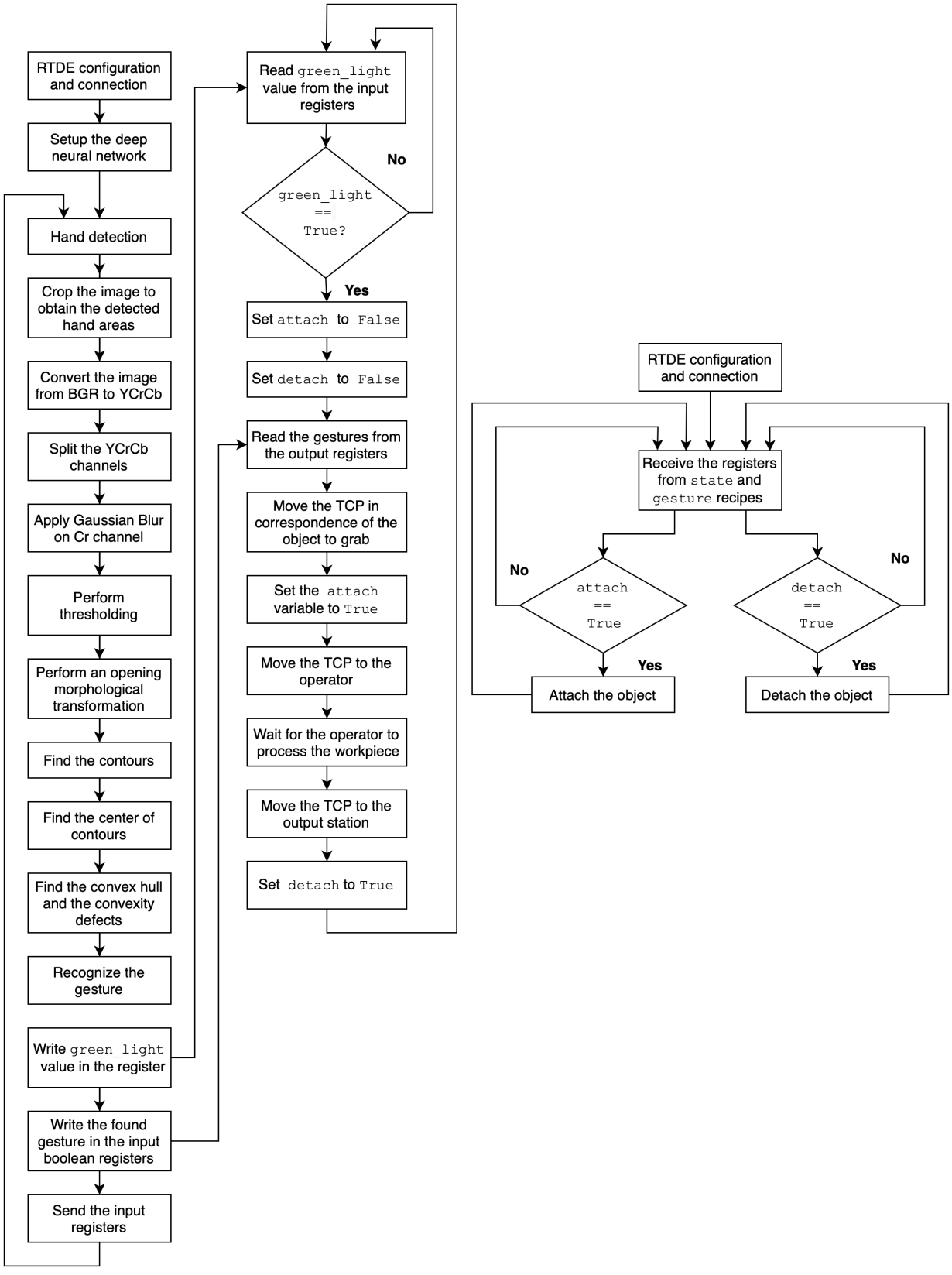

Figure 5.31: Flow chart of the algorithm for the case of study n.2. From left to right, the PC, robot and RoboDK programs.

each finger, a basic gesture recognizer can be realized. The gesture is first encoded in a string which contains the word of the performed gesture (see Listing 5.26).

Recipies	Name of the variable to synchronize	Types of the variable to synchronize
State	output_bit_register_72	BOOL
	output_bit_register_73	BOOL
Gesture	input_bit_register_65	BOOL
	input_bit_register_66	BOOL
	input_bit_register_67	BOOL
	input_bit_register_68	BOOL
	input_bit_register_69	BOOL
	input_bit_register_70	BOOL
	input_bit_register_71	BOOL
Watchdog	input_int_register_0	INT32

Table 5.2: Recipes with names and types of data to synchronize

```
if n_finger==5 and n_angle==4:
  gesture = 'Five'
elif n_finger in (4,5) and n_angle==3:
  gesture = 'Four'
elif n_finger in (3,4) and n_angle==2:
  gesture = 'Three'
elif n_finger in (2,3) and n_angle==1:
  gesture = 'Two'
elif n_finger in (1,2) and n_angle in (0,1):
  gesture = 'One'
else:
  gesture = 'None'

gesture_list.append(gesture)
cv2.putText(image_extract_3ch,gesture,(20,115), cv2.FONT_HERSHEY_SIMPLEX, 4,
    (0, 0, 255), 7, cv2.LINE_AA)
```

Listing 5.26: Five number gesture recognition algorithm

The intervals set in the number of fingers and in the number of detected angles in the `if` statements is inserted in order to make more robust the algorithm in the case in which the image has a bad quality and some parts of the hand, such as the wrist, are mistaken for fingertips. Figures 5.32 to 5.41 shows the gestures and their recognition.

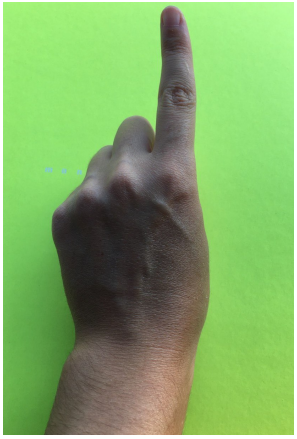

Figure 5.32: Gesture 'One'.

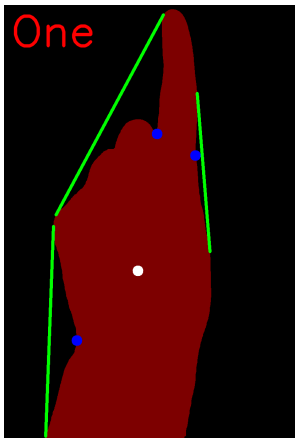

Figure 5.33: Recognition of gesture 'One'.

Figure 5.34: Gesture 'Two'.

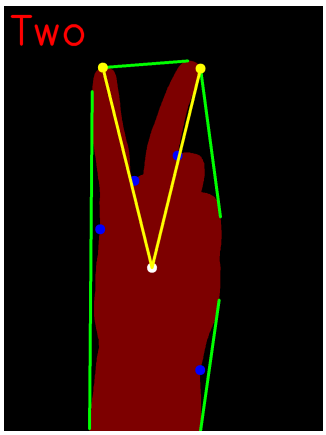

Figure 5.35: Recognition of gesture 'Two'.

Figure 5.36: Gesture 'Three'.

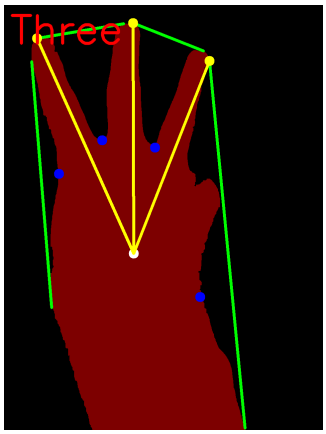

Figure 5.37: Recognition of gesture 'Three'.

The performed gesture is sent to the simulated robot using registers available through the RTDE connection. In particular, according to the string which indicates the gesture to perform, Listing 5.27 encodes it into a binary value stores in the input registers.

```
if final_gesture == 'Five':
  gesture.input_bit_register_70 = 1
  gesture.input_bit_register_69 = 0
  gesture.input_bit_register_68 = 0
  gesture.input_bit_register_67 = 0
  gesture.input_bit_register_66 = 0
  gesture.input_bit_register_65 = 0

elif final_gesture == 'Four':
  gesture.input_bit_register_70 = 0
  gesture.input_bit_register_69 = 1
  gesture.input_bit_register_68 = 0
  gesture.input_bit_register_67 = 0
  gesture.input_bit_register_66 = 0
  gesture.input_bit_register_65 = 0

elif final_gesture == 'Three':
  gesture.input_bit_register_70 = 0
  gesture.input_bit_register_69 = 0
  gesture.input_bit_register_68 = 1
  gesture.input_bit_register_67 = 0
  gesture.input_bit_register_66 = 0
   gesture.input_bit_register_65 = 0

 elif final_gesture == 'Two':
  gesture.input_bit_register_70 = 0
  gesture.input_bit_register_69 = 0
  gesture.input_bit_register_68 = 0
  gesture.input_bit_register_67 = 1
  gesture.input_bit_register_66 = 0
  gesture.input_bit_register_65 = 0

elif final_gesture == 'One':
  gesture.input_bit_register_70 = 0
  gesture.input_bit_register_69 = 0
  gesture.input_bit_register_68 = 0
  gesture.input_bit_register_67 = 0
  gesture.input_bit_register_66 = 1
  gesture.input_bit_register_65 = 0

elif final_gesture == 'None':
  gesture.input_bit_register_70 = 0
  gesture.input_bit_register_69 = 0
  gesture.input_bit_register_68 = 0
  gesture.input_bit_register_67 = 0
  gesture.input_bit_register_66 = 0
  gesture.input_bit_register_65 = 0
```

Listing 5.27: Setting input bit registers of the second case of study to send the performed gesture to the simulated robot

Figure 5.38: Gesture 'Four'.

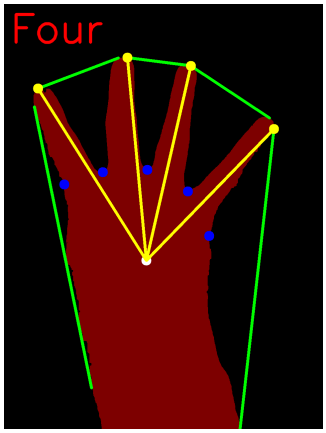

Figure 5.39: Recognition of gesture 'Four'.

Figure 5.40: Gesture 'Five'.

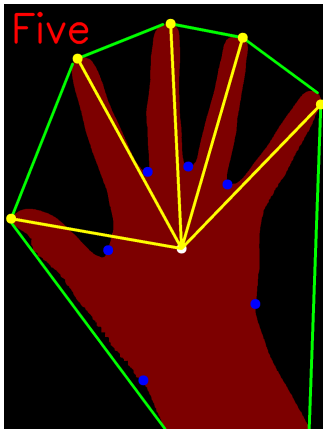

Figure 5.41: Recognition of gesture 'Five'.

5.3.3 Polyscope and RoboDK

The robot program in Polyscope allows the robot to perform the movement: to each movement a register is associated which is read by the robot program and each movement corresponds to a sequence of operations that the robot arm has to perform.

The flow chart of the robot program is shown in the middle column of the Figure 5.31. The variable `green_light` stores the value of a register which if `True` indicates that the performed gesture has already been interpreted by the algorithm and encoded into the registers. The `attach` and `detach` variables allow to communicate to the RoboDK program whether an object has to be grabbed or dropped.

All the movements which the robot performs are programmed using Polyscope. To move fast from a TCP pose to another MoveJ is used, while for end movements such as the picking or the placing of a workpiece MoveL is used.

RoboDK replicates the robot movements performed in Polyscope and allows to visualize a more realistic environment with an operator, workpieces and workbenches. Also, a Python script runs inside the RoboDK environment. In this script, the RTDE connection is established and this allows the communication and the data exchange between Polyscope and RoboDK. The registers which corresponds to `attach` and `detach` variables are written in Polyscope and are accessed y RoboDK. These variables are properly set in order to perform grabbing the piece when the end-effector is enough close to the workpiece and dropping the piece when the end-effector can place the object on the workbench surface. The Figure 5.42 shows the simulation environment.

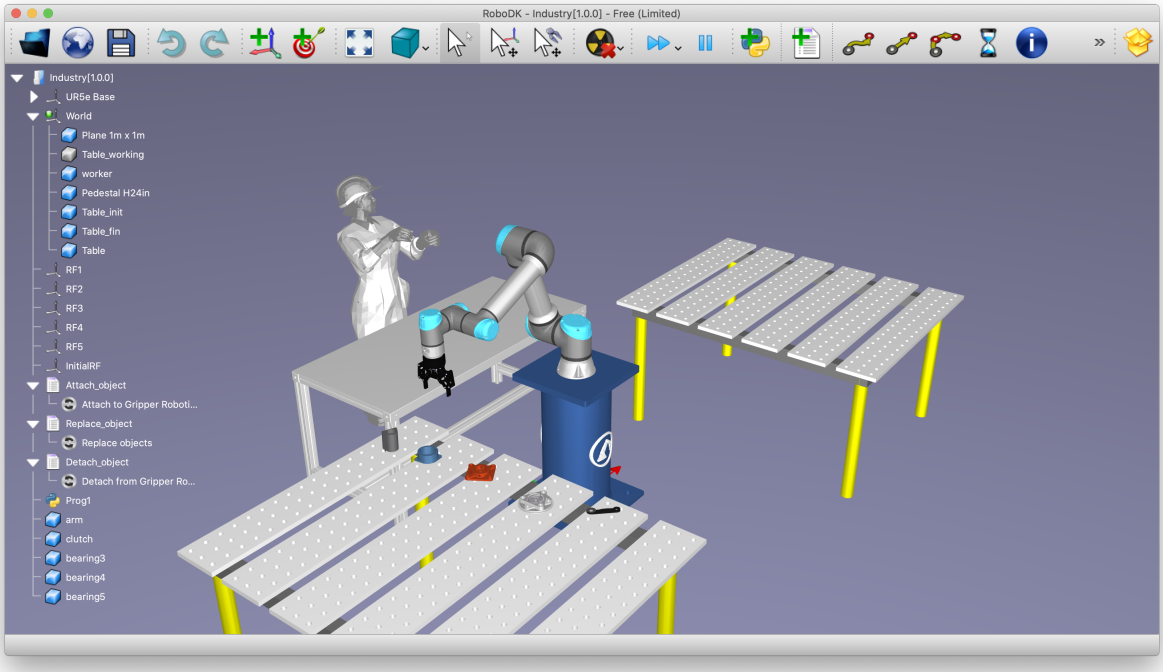

Figure 5.42: Simulation of the third case of study in RoboDK.

5.4 Gesture recognition in an hospital

Patients might be not self-sufficient and they might stay on an hospital bed for days or months without being able to use arms or the legs. In these situations, giving to the patient a certain degree of autonomy is very important. A collaborative robot could help the patient to get the objects he/she needs without getting help from a health care worker.

Figure 5.43 shows a complete overview of the programs which run in parallel on the PC, Polyscope and RoboDK and which will be illustrated in the following sections.

5.4.1 RTDE and data exchange

The RTDE setup for this case of study is similar to the one presented for the second case of study, the only difference being in the recipe used for encoding gestures (see Table 5.3). In particular, in this case only three input registers are used since only three gestures can be recognized. The function of the state and watchdog recipes is explained in Section 5.3.1 and

Recipies	**Name of the variable to synchronize**	**Types of the variable to synchronize**
State	output_bit_register_72	BOOL
	output_bit_register_73	BOOL
Gesture	input_bit_register_65	BOOL
	input_bit_register_66	BOOL
	input_bit_register_67	BOOL
Watchdog	input_int_register_0	INT32

Table 5.3: Recipes with names and types of data to synchronize

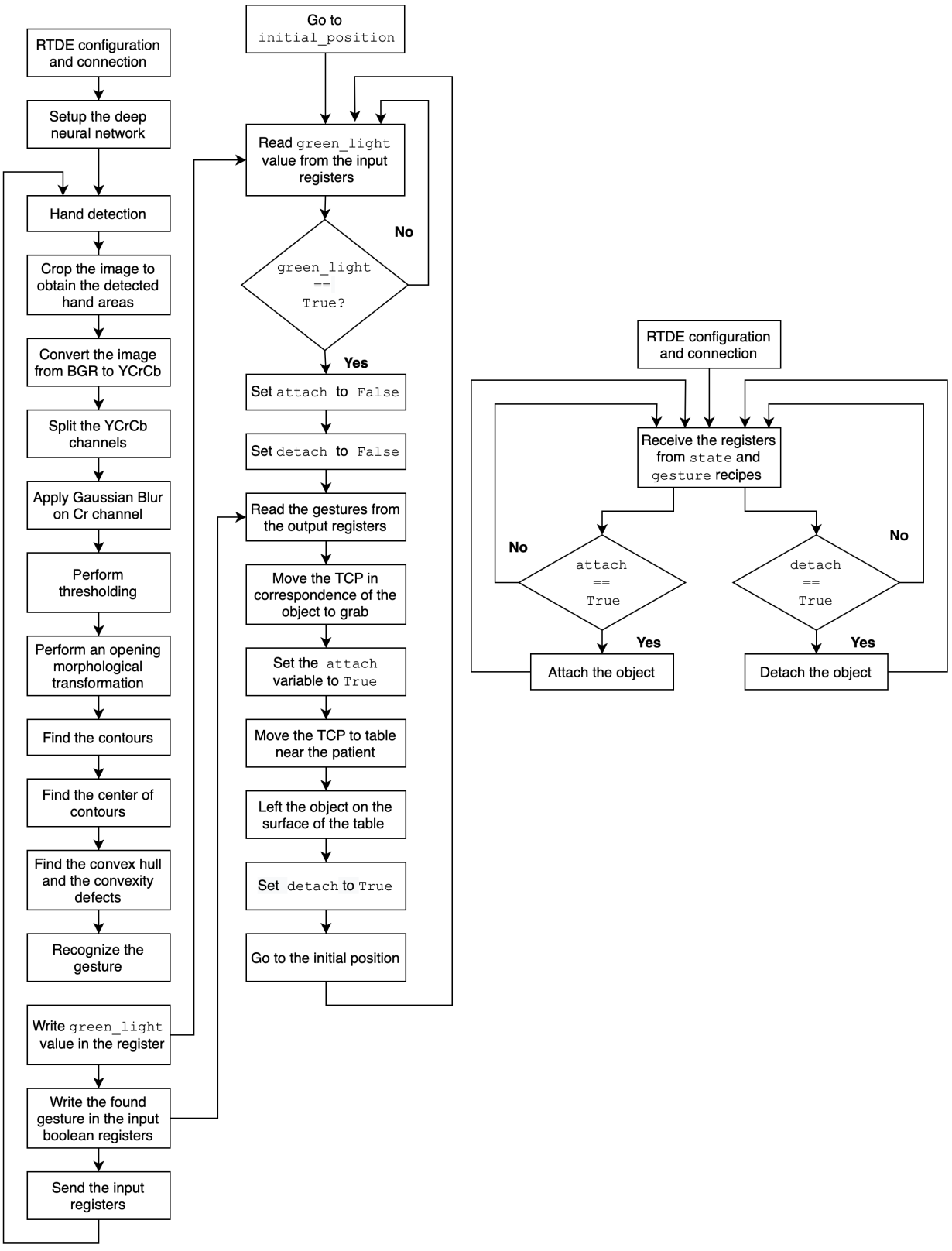

Figure 5.43: Flow chart of the algorithm for the case of study n.3. From left to right, the PC, robot and RoboDK programs.

5.1.3 respectively. The `input_bit_register_65` is associated to the getting the telephone, the `input_bit_register_66` to the getting the drugs and the `input_bit_register_67` to the getting the glass.

5.4.2 Three guidance gestures recognition

Similarly to the second case of study, the analysis of the number of detected fingers and the angles allows to the algorithm to recognize the gesture performed by the patient. The string encodes the name of the gesture which the patient performed. The algorithm recognizes three different gestures, corresponding to three different actions: getting the telephone, the medicines and drinks (see Listing 5.28).

```
if n_finger in (2,3) and n_angle==1:
  gesture = 'Telephone'
elif n_finger in (3,4) and n_angle in (2,4):
  gesture = 'Meds'
elif n_finger in (1,2) and n_angle in(0,1):
  gesture = 'Drink'
else:
  gesture = 'None'

gesture_list.append(gesture)
cv2.putText(image_extract_3ch,gesture,(20,115), cv2.FONT_HERSHEY_SIMPLEX, 4,
    (0, 0, 255), 7, cv2.LINE_AA)
```

Listing 5.28: Setting input bit registers of the third case of study to send the performed gesture to Polyscope

Figures from 5.44 to 5.49 show the gestures and their recognition. Similarly to the code presented in the Section 5.3.3, the gestures are encoded in registers to be sent to the simulated robot. The code which performs this operation is shown in Listing 5.29.

```
if final_gesture == 'Telephone':
  gesture.input_bit_register_65 = 1
  gesture.input_bit_register_66 = 0
  gesture.input_bit_register_67 = 0

elif final_gesture == 'Meds':
  gesture.input_bit_register_65 = 0
  gesture.input_bit_register_66 = 1
  gesture.input_bit_register_67 = 0

elif final_gesture == 'Drink':
  gesture.input_bit_register_65 = 0
  gesture.input_bit_register_66 = 0
  gesture.input_bit_register_67 = 1

elif final_gesture == 'None':
  gesture.input_bit_register_65 = 0
  gesture.input_bit_register_66 = 0
  gesture.input_bit_register_67 = 0
```

Listing 5.29: Three gesture recognition algorithm

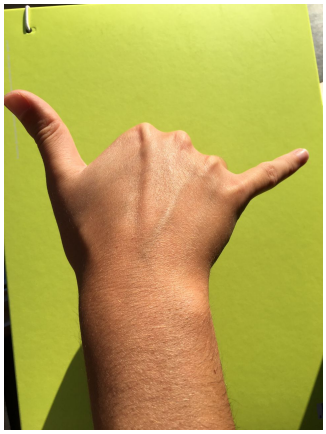

Figure 5.44: Gesture 'Telephone'.

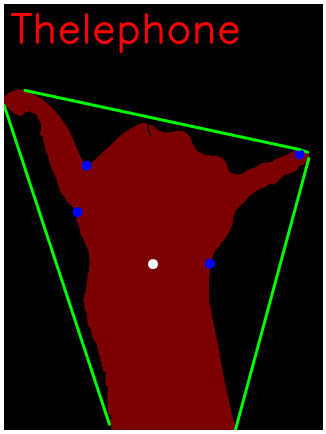

Figure 5.45: Recognition of gesture 'Telephone'.

Figure 5.46: Gesture 'Medicine'.

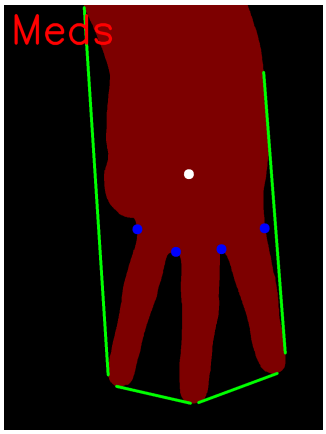

Figure 5.47: Recognition of gesture 'Medicine'.

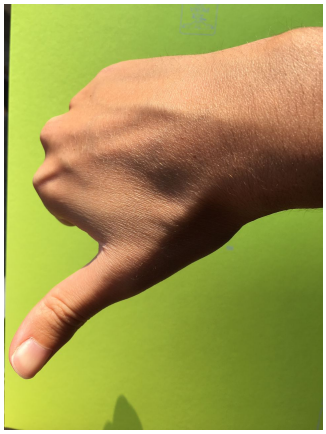

Figure 5.48: Gesture 'Drink'.

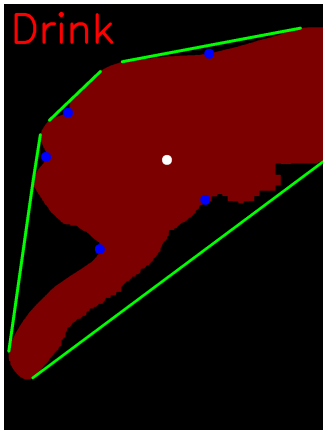

Figure 5.49: Recognition of gesture 'Drink'.

5.4.3 Polyscope and RoboDK

The robot program, which runs in Polyscope, performs the movement of the robotic arm. To each gesture corresponds a sequence of movement that the robot has to perform to grab the object and hands it to the patient. After performing the complete sequence of movements, the robot returns to its initial position. The explanation of the meaning of the variables in the middle column of Figure 5.43 can be found in Section 5.3.3.

Also in this case, as explained in Section 5.3.3, RoboDK allows to visualize a more realistic simulation of the environment around the patient. The hospital bed, bedside table and medicine cabinet provide a more realistic context. As seen previously, RoboDK is necessary to simulate the grabbing and the dropping of the object. The Figure 5.50 shows the simulation environment.

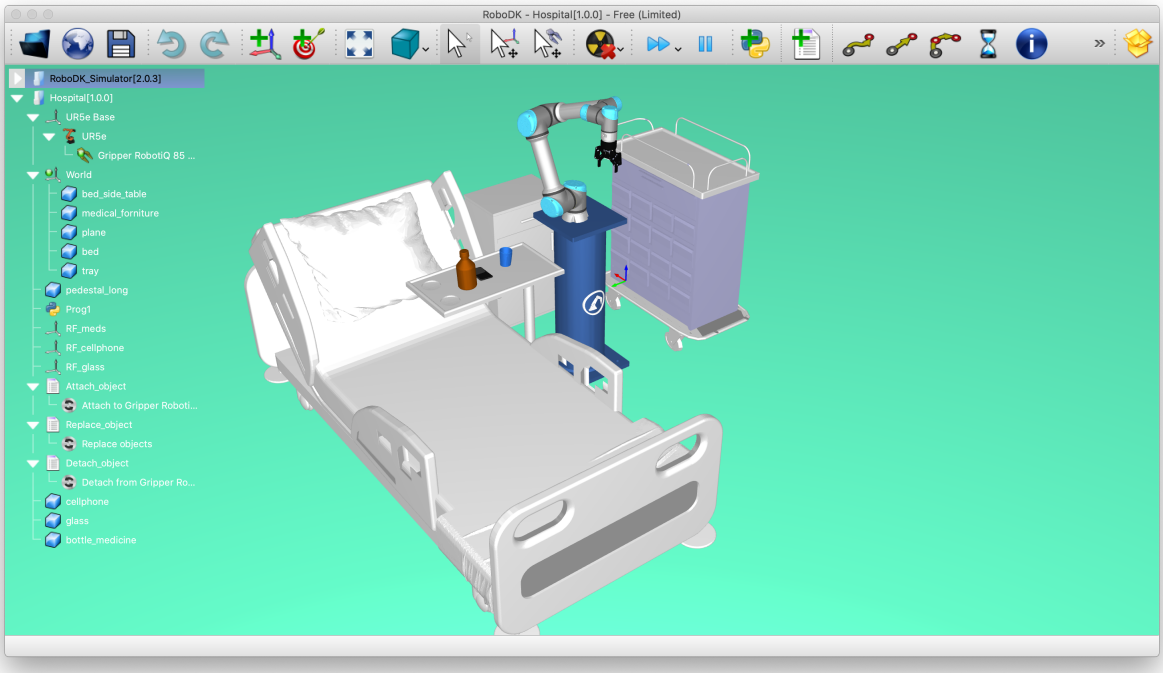

Figure 5.50: Simulation of the third case of study in RoboDK.

Chapter 6

Conclusion

In this thesis, Artificial Intelligence and Computer Vision techniques were explored to enable the interaction between collaborative robots and humans in a variety of contexts (television studio, manufacturing plant and hospital bedroom).

In the first case of study, the interaction between human and collaborative robot is based on a face tracking algorithm: a robotic cameraman follows the face of the television presenter thanks to a face-tracking algorithm and shoots the video. To do this, a high level algorithm in *Python* language, which runs on a dedicated computer, is developed. This algorithm also enables the communication between the robot and the computer in which the face-tracking algorithm is implemented. Its main output is the pose which the robot has to reach and which is sent to the robot through the *Real-Time Data Exchange* interface. The robot's movement is simulated in *Polyscope*.

In the second and the third cases of study, the interaction between humans and robots is performed through gesture recognition. A human performs a gesture in front of a camera, the frame is elaborated at first detecting the hand and cropping the frame around the *Region of Interest* which contains the hand, and then applying processing techniques to this image which allow to recognize the number of fingers detected and the angles between them. Once the gesture is recognized, it is written on the corresponding register using boolean values and the registers are sent to robot using the *Real-Time Data Exchange* interface. In this way, the action to perform is sent to *Polyscope*. To analyze the correctness of the actions performed by the robot, such as the attaching and the detaching of the gripper, and to simulate the objects in a realistic environment in which several types of objects are present, *RoboDK* is used. In both cases of study, to each gesture corresponds an action to perform: in the manufacturing plant, according to the performed gesture, the robot grabs a workpiece from the input station, brings it to the operator and waits until he finishes to process it. After it is processed, the workpiece is brought to the output station. In the hospital bedroom case, the object (medicine, telephone or glass of water) is taken by the robot and left in front of the patient.

The simulation of the three cases of study on both Polyscope and RoboDK shows promising outcomes and, thus, they might be implemented on a real context after careful testing on a UR arm. For example, in a real situation, to avoid any kind of dangerous situation, the appropriate safety configuration (see Section 3.1.1) has to be chosen for each case after an appropriate risk evaluation. In particular, after the identification of the safety area, the safety mode has to be set: in the television studio and in the manufacturing plant, the *reduce mode* (i.e. the robot moves slower when a person is nearby) would be used; in the hospital case, the *safeguard stop and continue* (i.e. the robot stops when a person is nearby) might be more appropriate for the patient safety.

Moreover, the first case of study might receive a new impulse from testing: using a camera which is mounted on the end-effector, the pose of the TCP might be computed not only based

the translation, but also considering the orientation. In this case, the camera would not be fixed (as the webcam of the computer), but it could also rotate with the end-effector. In this way, a more sophisticated robotic cameramen, able to follow the television presenter also in positions which are difficult to reach using translation only, could be designed.

Considering the second and the third cases of study, a more sophisticated gesture recognition algorithm could be implemented in order to obtain a more extensive library of gestures to perform. An idea could be to directly use a classification algorithm, training a neural network to recognize the gestures. In this way, the recognition might be more accurate and robust.

Bibliography

[1] Gesture recognition based on image processing and machine learning, https://github.com/zzeitt/gesture-recognition.

[2] A/S, U. R. Universal robots website.

[3] A/S, U. R. The urscript programming language, January 2020.

[4] Bambach, S., Lee, S., Crandall, D. J., and Yu, C. Lending a hand: Detecting hands and recognizing activities in complex egocentric interactions. In *Proceedings of the IEEE International Conference on Computer Vision* (2015), pp. 1949–1957.

[5] Bartneck, C., Belpaeme, T., Eyssel, F., Kanda, T., Keijsers, M., and Šabanović, S. *Human-Robot interaction: An introduction*. Cambridge University Press, 2020.

[6] Boccanfuso, L., and O'Kane, J. M. Charlie: An adaptive robot design with hand and face tracking for use in autism therapy. *International journal of social robotics 3*, 4 (2011), 337–347.

[7] Bona, B. *Dynamic modelling of mechatronic systems*. Celid, 2013.

[8] Bradski, G., and Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[9] Brethes, L., Menezes, P., Lerasle, F., and Hayet, J. Face tracking and hand gesture recognition for human-robot interaction. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004* (2004), vol. 2, IEEE, pp. 1901–1906.

[10] Colgate, J. E., and Peshkin, M. A. Cobots, Sept. 14 1999. US Patent 5,952,796.

[11] Control Systems Technology Group Wiki, Eindhoven University of Technology (TU/e). Convexity defects, 2014. [Online; accessed August-2020].

[12] Dhawan, A., and Honrao, V. Implementation of hand detection based techniques for human computer interaction. *arXiv preprint arXiv:1312.7560* (2013).

[13] Dibia, V., and using Neural, R.-t. H.-D. Networks (ssd) on tensorflow,(2017), github repository.

[14] for Standardization (ISO), I. O. Iso/ts 15066:2016 robots and robotic devices - collaborative robots.

[15] Freund, Y., Schapire, R. E., et al. Experiments with a new boosting algorithm. In *icml* (1996), vol. 96, Citeseer, pp. 148–156.

[16] Géron, A. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[17] Girshick, R. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision* (2015), pp. 1440–1448.

[18] Girshick, R., Donahue, J., Darrell, T., and Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2014), pp. 580–587.

[19] Goodrich, M. A., and Schultz, A. C. *Human-robot interaction: a survey*. Now Publishers Inc, 2008.

[20] Hasanuzzaman, M., and Ueno, H. Face and gesture recognition for human-robot interaction. *Face Recognition* (2007).

[21] Hawkins, K. P. Analytic inverse kinematics for the universal robots ur-5/ur-10 arms. Tech. rep., Georgia Institute of Technology, 2013.

[22] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[23] Hentout, A., Aouache, M., Maoudj, A., and Akli, I. Human–robot interaction in industrial collaborative robotics: a literature review of the decade 2008–2017. *Advanced Robotics 33*, 15-16 (2019), 764–799.

[24] Homma, K., and Takenaka, E.-i. An image processing method for feature extraction of space-occupying lesions. *Journal of nuclear medicine: official publication, Society of Nuclear Medicine 26*, 12 (1985), 1472–1477.

[25] Hulstaert, L. A beginner's guide to object detection, 2018. [Online; accessed 30-August-2020].

[26] Ikeuchi, K. *Computer vision: A reference guide*. Springer Publishing Company, Incorporated, 2014.

[27] Jakubiec, B. Application of simulation models for programming of robots. In *Proceedings of the International Scientific Conference. Volume V* (2018), vol. 283, p. 292.

[28] Jensen, O. H. Implementing the viola-jones face detection algorithm. Master's thesis, Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2008.

[29] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia* (2014), pp. 675–678.

[30] Jost, C., Le Pévédic, B., Belpaeme, T., Bethel, C., Chrysostomou, D., Crook, N., Grandgeorge, M., and Mirnig, N. *Human-Robot Interaction: Evaluation Methods and Their Standardization*, vol. 12. Springer Nature, 2020.

[31] Kovac, J., Peer, P., and Solina, F. Human skin color clustering for face detection. In *In International Conference on Computer as a Tool. EUROCON2003* (2003), vol. 2, IEEE.

[32] Lienhart, R., and Maydt, J. An extended set of haar-like features for rapid object detection. In *Proceedings. international conference on image processing* (2002), vol. 1, IEEE, pp. I–I.

[33] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. Ssd: Single shot multibox detector. In *European conference on computer vision* (2016), Springer, pp. 21–37.

[34] Loresco, P., and Bandala, A. Human gesture recognition using computer vision for robot navigation. In *IEEE* (2018).

[35] Moeslund, T. B. *Introduction to video and image processing: Building real systems and applications*. Springer Science & Business Media, 2012.

[36] Nielsen, A. M. *Neural Networks and Deep Learning*. Determination Press, 2015.

[37] Oña, E. D., Garcia-Haro, J. M., Jardón, A., and Balaguer, C. Robotics in health care: Perspectives of robot-aided interventions in clinical practice for rehabilitation of upper limbs. *Applied sciences 9*, 13 (2019), 2586.

[38] OpenCV. Opencv deep learning module samples.

[39] Peshkin, M. A., Colgate, J. E., Wannasuphoprasit, W., Moore, C. A., Gillespie, R. B., and Akella, P. Cobot architecture. *IEEE Transactions on Robotics and Automation 17*, 4 (2001), 377–390.

[40] Pessanha, F., and Correia, P. Hand-pose recognition, June 2019.

[41] Prince, S. J. *Computer vision: models, learning, and inference*. Cambridge University Press, 2012.

[42] Putro, M. D., and Jo, K.-H. Real-time face tracking for human-robot interaction. In *2018 International Conference on Information and Communication Technology Robotics (ICT-ROBOT)* (2018), IEEE, pp. 1–4.

[43] Ren, S., He, K., Girshick, R., and Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems* (2015), pp. 91–99.

[44] RoboDK. Simulate robot applications - program any industrial robot with one simulation environment.

[45] Robotiq. 2f-85 and 2f-140 grippers. *Online datasheet* (2019).

[46] Rockafellar, R. T. *Convex analysis*. Princeton university press, 1970.

[47] Siciliano, B., Sciavicco, L., Villani, L., and Oriolo, G. *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010.

[48] Sick. Safe robotics transfer: Productivity restarts automatically, December 2016.

[49] Szeliski, R. *Introduction*. Springer London, London, 2011, pp. 1–25.

[50] Topalidou-Kyniazopoulou, A. Motion planning strategy for a 6-dofs robotic arm in a controlled environment.

[51] tutorials, O. P. Morphological transformations.

[52] Van Oosterwyck, N. *Real Time Human-Robot Interactions and Speed Control of a Robotic Arm for Collaborative Operations*. PhD thesis, Ph. D. dissertation, 05 2018, 2018.

[53] Villán, A. F. *Mastering OpenCV 4 with Python: a practical guide covering topics from image processing, augmented reality to deep learning with OpenCV 4 and Python 3.7*. Packt Publishing Ltd, 2019.

[54] Wikipedia contributors. Barns grand tetons ycbcr separation, 2006. [Online; accessed August-2020].

[55] Wikipedia contributors. Convexity defects, 2020. [Online; accessed September-2020].

[56] Xu, X., Xu, S., Jin, L., and Song, E. Characteristic analysis of otsu threshold and its applications. *Pattern recognition letters 32*, 7 (2011), 956–961.