POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Instance Segmentation and Visual Servoing for Apple Harvesting

Supervisors:

Prof. Marcello CHIABERGE

PhD. Mauro MARTINI

Dott. Marco AMBROSIO

Dott. Alessandro NAVONE

Candidate:

Adriana DI TERLIZZI

Academic Year 2023/2024

Abstract

In recent years, the rising global population and the growing demand for food, coupled with a shortage of human laborers, have placed immense pressure on the agricultural sector to revolutionize farming operations through advanced technology, to produce more food, more efficiently and sustainably. In this context, autonomous harvesting robots have emerged as a promising solution to these challenges.
This thesis focuses on designing a visual servo for apple harvesting using the Kinova Gen 3 Lite, a 6 Degrees of freedom (DoF) manipulator, equipped with an Intel RealSense D435i camera mounted on its end effector. The objective is to enable the robot to accurately identify the target apple and autonomously guide the robot tool toward it for successful grasping.
To address the problems, the proposed solution comprises two main modules: one dedicated to visual perception and another focused on manipulator control. Instance Segmentation is selected as a precise and effective strategy to cope with target recognition and localization of individual apples. A YOLOv8 Convolutional Neural Network (CNN) has been trained and fine-tuned to perform this task. Meanwhile, an Image-Based Visual Servo (IBVS) is employed to control the robot's motion efficiently. A comprehensive and integrated pipeline has been developed to evaluate the individual performances of the two modules and their combined efficacy. The Kinova robot operates on a stationary basis with static targets. By real-time inferencing the data stream captured by the RGB-D sensor, the YOLOv8 CNN model detects and segments all apple instances. A selection policy is then applied to choose the target apple, whose visual features are exploited by the IBVS for feedback control. Through direct error computation in the image space, this closed-loop system continuously adjusts the camera motion, thus the end effector motion, navigating the robot tool to the desired pose. Upon proximity to the target, an open-loop approach is executed to facilitate apple grasping.
The proposed architecture is fully implemented in Robot Operating System 2 (ROS2), with Python as the primary programming language. Extensive experiments were conducted using the real robot and mockup apples at PIC4SeR (PoliTo Interdepartmental Centre for Service Robotics). The results show effective apple instance segmentation and localization by the network, while the IBVS controller correctly drives the end effector toward the target.

Acknowledgements

I am grateful to my supervisor, Prof. Marcello Chiaberge, and all my co supervisors from the PIC4SeR team — Marco, Mauro, Alessandro and Luigi — for giving me the opportunity to work on this project, and for assisting and supporting me throughout the thesis process.

Table of Contents

List of Tables

List of Figures

Acronyms

SW
Software

HW
Hardware

AI
artificial intelligence

VS
Visual Servoing

IBVS
Image Based Visual Servo

PBVS
Position Based Visual Servo

DL
Deep Learning

ML
Machine Learning

CNN
Convolutional Neural Network

RF
Reference Frame

DOF

Degrees Of Freedom

NN

Neural Network

ANN

Artificial Neural Network

TLU

Threshold Logic Unit

MLP

Multilinear Perceptron

ROS

Robot Operating System

GUI

Graphical User Interface

API

Application Programming Interface

PC

Personal Computer

LAN

Local Area Network

GPS

Global Positioning System

RTUs

Robotic Transport Units

RGB

Red Green Blue

LiDAR

Light Detection and Ranging

IS

Instance Segmentation

MC

Multi Classification

IOU

Intersection Over Union

SORT

Simple Online and Realtime Tracking

Chapter 1

Introduction

In recent years, the increasing global population and the growing demand for food, combined with a shortage of human labor, have exerted considerable pressure on the agricultural sector to modernize farming operations through advanced technological solutions. The goal is to produce more food, in a more efficient and sustainable manner. Within this framework, autonomous harvesting robots have emerged as a promising solution to these challenges.

Particularly, autonomous harvesting manipulators are engineered to transform the harvesting process into a manageable, trackable and customizable operation. Research in this domain encompasses the development of sophisticated sensors and algorithms for fruit detection, advanced path planning strategies to approach targets, innovative obstacle avoidance techniques and refined methods for grasping ripe fruit without causing damage.

The idea of the thesis originated at PIC4SeR (PoliTo Interdepartmental Centre for Service Robotics), within a broader initiative aimed at advancing service robotics to create cutting-edge solutions across various domains, including precision agriculture, smart urban environments, healthcare, cultural heritage preservation and space exploration.

Objective of the project

This thesis focuses on designing a Visual Servo for Apple Harvesting using the Kinova Gen 3 Lite, a 6 Degrees of freedom (DoF) manipulator, equipped with an Intel RealSense D435i camera mounted on its end effector. The primary objective is to implement a robust apple identification algorithm, to enhance the robot with visual perception of the target fruit. Then, the second goal is enabling the robot tool to autonomously approach the identified apple for completing successful

grasping, by means of a visual controller, exploiting the visual data coming from the RGBD sensor. The final aim is to create a comprehensive, integrated system that combines the Vision and Control modules. The complete architecture will be thoroughly evaluated to assess the individual performances of each module, and their combined efficacy in executing successful autonomous harvesting operations.

Organization of the thesis work

An overview of the thesis structure is provided, offering a brief description of the content covered in each chapter.

- *Chapter 2* introduces the role of robotics in agriculture, with a particular emphasis on autonomous harvesting systems. It presents a comprehensive review of state-of-the-art apple harvesting robots, highlighting innovative designs from both academic research and startup initiatives.
- *Chapter 3* presents the theoretical framework of the system's key technologies, covering the implemented visual controller, the Image-Based Visual Servo (IBVS) approach, as well as the underlying principles of Convolutional Neural Networks and YOLOv8 architecture.
- *Chapter 4* offers an in-depth exposition of the proposed system architecture, comprising the Vision model and the Visual Controller, detailing how they work to achieve the desired objective.
- *Chapter 5* provides an overview of the software and hardware tools employed in the system's development, simulation and testing phases.
- *Chapter 6* shows the outcomes of extensive experimental tests carried out in a realistic environment, providing an analysis of the system's performance.
- *Chapter 7* concludes the thesis by summarizing the key findings and exploring potential directions for future work.
- The *Appendix* contains additional insights on topics referenced throughout the thesis, intended to assist the reader in gaining a deeper understanding.

Chapter 2

State of the art

As the global population continues to rise, the demand for food production has surged correspondingly. This growing demand places immense pressure on the agricultural sector to produce more food, more efficiently, and sustainably. Traditional farming methods, reliant on manual labor and extensive use of resources, are increasingly unable to meet this challenge. Consequently, the agricultural industry is turning to advanced technologies, particularly robotics, to revolutionize farming practices.
This chapter explores the current state of smart solutions in agricultural robotics, with a particular focus on robotic systems designed for the harvesting task.

2.1 Introduction to Agricultural Robotics

In everyday life, robotic solutions are increasingly prevalent across various sectors, from smart factories and service robotics to autonomous driving. This technological advancement is also making meaningful inroads into agriculture.
The integration of robots into precision agriculture have boosted efficiency and versatility, by guaranteeing a continuous work also in labor-intensive tasks; as well as optimized the use of resources like water and fertilizers, promoting sustainability.
The farming robots are mainly classified according to the task they accomplish [1]:

- **Seeding**: These machines automate planting seeds and crops. By using GPS, these robots optimize both depth and space for each planted seed.
- **Fertilizing**: This role consists of spreading the fertilizer over the entire cultivated field. The manual work, as well as being intense and difficult, could lead to irregular spreading. Robots instead act on specific areas: they directly fertilize the soil or the plant. To achieve so, the robots have proper sensors and mapping technologies, enabling them to navigate and localize the areas of

interest.
Some robots deploy pneumatic systems to inject fertilizer pellets into the soil, while others apply liquid fertilizer by hitting the plants.

- **Weeding and Pest Control**: These units should be equipped with some object recognition technologies to first identify the weeds to get rid of and with some precision tools to remove the weeds from the soil. Since weeds could be anywhere, robots should be able to extend or shorten their reach: that's why mobile and articulated robots are well-suited for this task. To increase the efficiency of these systems and reduce the operational space of the machines, the gathered data on the presence of weeds in past weeding operations is important. In this way, it is estimated where and when they are most likely to reappear.

- **Monitoring and Scouting**: Drones or ground robotic machines are deployed to collect information on plant vitality, soil hydration, and additional key agricultural indicators.

- **Harvesting**: These robots aim at picking and placing fruits and vegetables. They employ sensors and cameras to identify when it's time to harvest the fruits and they use arms or other precision tools to delicately collect them without harming the yield.

2.1.1 Insight on robot models involved in agriculture

As we have described the agricultural operations being automated, let's now look into the robot models currently involved in these processes [1]:

- **Six-Axis Robots**: This kind of robot features a flexible arm with multiple joints, enabling movement in several directions and access to a broad range of positions. This extensive articulation and the ability to navigate tight spaces make it effective in operations requiring intricate movements, like the harvesting of fruits and vegetables.
  Six-Axis Robot's huge reach enables it to extend over planters or other obstacles to pick or spray crops effectively.
  A key aspect of Six-Axis Robots is the presence of advanced sensors, including vision systems, guiding them on picking fruits ripe enough to be grasped.
  Since they have limited mobility on their own, they are most likely to be used on robotic transport units (RTUs) or integrated with mobile robots.

- **Mobile Robots**: Designed with wheels or tracks, they move through fields and other outdoor settings with ease. Beyond merely transporting other robots from one location to another, they also perform crop monitoring. Outfitted

with cameras and other sensors, these robots gather valuable data on plant health, soil moisture, helping farmers in irrigation, fertilization and various aspects of crop management.

- **Autonomous Tractors**: They are valuable tools for planting, fertilizing and spraying. Equipped with GPS and mapping technologies, these tractors navigate fields at steady speeds and maintain uniform application rates, resulting in more consistent crop growth and higher yields. The risk of accidents and crop is way reduced because they can avoid obstacles. Because of their continuous work, they can boost efficiency and cut labor costs for: soil preparation, tilling, seeding and harvesting.

2.2 Autonomous robots for harvesting

In recent years, significant advancements have been made in the development of autonomous robots for harvesting a wide range of crops, including apples, strawberries, grapes and more. These robots are engineered to transform the harvesting process into a manageable, trackable, and tailorable operation; to acquire, interpret, and evaluate individual information of the target fruits (maturation periods, harvest time, fruit position and imperfections); to tackle the challenges presented by the agricultural environment (such as branch arrangements, foliar obstruction, and disease infection). By integrating this information with plant growth patterns and varietal characteristics, the harvesting systems provide valuable insights to guide agricultural decision-makers in optimizing their operations.
In the context of autonomous harvesting robots, two harvesting approaches have been developed: selective harvesting and bulk harvesting [2]. The first method uses a mobile basis on which robotic arms, equipped with a tool for picking and a camera for target detection, are mounted.
The second method involves shaking the fruit tree so that the fall of target fruits is caused. However, since this approach exhibits various cons, including the harming of fruits and canopies and the harvesting of both ripe and immature fruits, it it less common than the selective harvesting approach [2].
In the following, a dive into the system architecture of robots for selective harvesting will be presented.

2.2.1 System architecture of harvester robots

We can define harvesting robots as complex systems resulting from the integration of multiple subsystems [2], including:

- a mobile platform to navigate along the fields;

- a computer vision system aiming at recognizing the target and perceiving the environment;
- a control system to achieve the desired movement of the manipulator;
- a kind of box to place the picked fruits;
- one or more robotic arms or manipulators to pick the fruits while avoiding collisions;
- one or more grippers to grasp the fruit.

2.2.2 Fruit detection: sensors and algorithms

Before an apple can be located and grasped, a robotic harvester must first identify it accurately, even when faced with challenges such as variability in fruit size, shape and orientation, along with occlusion by leaves and branches, and changing lighting and weather.

Sensors

To identify the target and locate it, the harvesters are furnished with either 2D (two-dimensional) or 3D (three-dimensional) vision sensors.
Among the 2D sensors, we can mention: the RGB (red, green, blue colors) camera, the IR (infrared) sensor, spectral sensors, or a combination of any of them [3]. By applying traditional ML (machine learning) methods or DL (deep learning) algorithms, it's possible to identify the target fruit by analyzing the images recorded by the RGB camera (sometimes too sensitive to illumination). Spectral sensors can be used for getting both spectral and spatial information about the fruit by analyzing the different reflectance patterns at various wavelengths. Thermal sensors, capturing temperature data, help in distinguishing fruits from the background: as a matter of fact, fruits tend to absorb and emit more heat than the surrounding canopies.
For what concerns the 3D imaging sensors, we can refer to: stereo cameras, LiDAR (Light Detection and Ranging) sensors and RGB-D cameras. Stereo cameras capture images using two or more RGB cameras placed at a fixed distance apart, then combine these images to calculate depth through triangulation. While accurate, this process is slow and requires frequent calibration, limiting its real-time usability. LiDAR generates 2D or 3D point clouds by using pulsed laser reflections to gather spatial information. Although LiDAR can be integrated with RGB cameras, the data fusion is slow, and the high cost of precise LiDARs restricts their use. RGB-D cameras provide a real-time feed of RGB images blended with depth information, enabling superior localization accuracy, robustness, and computational efficiency at

a lower cost than LiDAR. This makes RGB-D cameras highly suitable for practical applications. However, their performance can decline at close distances or in extreme weather conditions [2].

ML and DL algorithms for fruit detection

Traditional vision algorithms can analyze and encode features like color, shape, and texture from images captured by sensors. These features are then processed using machine-learning (ML) classifiers to recognize and categorize objects. However, color-based methods often struggle with fruits that have similar colors to leaves, especially under varying lighting conditions, making them suitable only in controlled environments. To improve accuracy, other image features such as texture, light intensity, and edge detection are incorporated to enhance RGB image analysis. For example, Kang et al. [4, 5] proposed an algorithm that uses hierarchical multi-scale feature extraction of color and shape, followed by K-means clustering for classifying regions of interest. This approach, known as multiple feature-based detection, combines various features into a single data structure for object detection. Despite these advancements, there remains a gap in real-time detection performance regarding computational efficiency, accuracy and robustness under natural lighting. To address these limitations, deep learning (DL) based methods have been developed as a potential solution.
Among various deep learning approaches, convolutional neural networks (CNNs) stand out as a supervised method that utilizes convolution and back-propagation to extract target features, significantly enhancing the accuracy and generalization of algorithms. Although deep learning can be applied to diverse data types (such as depth, RGB, and infrared images, or their combinations), to achieve high detection accuracy, the training process is time-consuming and requires a large dataset of labeled images [2].

2.2.3 Methods for fruit grasping and end-effector innovations

The design of fruit detachment mechanisms and end effectors is crucial for the successful harvesting of crops: these components are responsible for gently removing fruits from plants without causing damage, while also adapting to various fruit types and sizes. Some detachment methods involve separating the stem from the branch by applying external force either directly or indirectly to the stem. Five categories of methods can be listed: stem cutting, stem pulling, fruit twisting, fruit pulling and vacuum suction. The stem cut and stem pull techniques involve performing these actions while holding the stem in place. In contrast, the fruit twist and fruit pull methods apply their respective forces directly to the fruit itself.

Vacuum-based methods, on the other hand, use suction to remove the fruit without making direct contact with either the stem or the fruit.
For what concerns the end effector design, the most adopted grippers are: the soft grippers (mimicking the soft, adaptive touch of human hands, minimizing the risk of bruising or damaging the fruit), the suction cups (using vacuum pressure to grip the fruit, effective for fruits with smooth surfaces), claw or pincer grippers (mechanical grippers designed to hold the fruit securely using a claw-like mechanism), hybrid designs (deriving from any combination of them) [2].

2.2.4 Obstacles avoidance approaches

The navigation to fruits in a cluttered and dynamic environment while avoiding obstacles is another major challenge of harvester robots. To address these issues, several approaches have been proposed, depending on the type of obstacles [2].

- **Soft obstacles**: To cope with soft obstacles (e.g., leaves and stems), the solutions involve temporarily clearing the way. Mechanical systems can be used to move leaves out of the way so that the fruit is visible to the camera (Harvest CROO Robotics, [6]). Alternatively, robots with dual (or more) arms can mimic human actions by pushing leaves aside.
- **Rigid obstacles**: Instead, to cope with rigid obstacles, such as branches or complex structures, manipulators with multiple degrees of freedom (DOF) and sophisticated path-planning algorithms are preferred. The number of DOF can vary among designs, with some using six (for example, [5]), seven, or even nine DOF. While more DOF can improve maneuverability, they also add complexity and might reduce overall picking efficiency. To manage this, vision algorithms are employed to identify and exclude fruits that are too difficult to reach, assuming that the remaining fruits can be grasped using straightforward paths.

2.2.5 Evaluation of harvester robots' performance

To assess the effectiveness of the fruit harvesting robots, three key performance metrics are used [7]:

1. Harvest Success Rate: this metric measures how many ripe fruits are successfully harvested out of the total number of ripe fruits in the canopy. It's important to note that this count includes fruits that may have been damaged during harvesting.
2. Cycle Time: this represents the average duration of a complete harvest cycle, covering all steps from fruit recognition and localization to path planning,

grasping, collection, and moving among fruits. Any time lost due to failed attempts is included if reported.

3. Fruit Damage Rate: this metric tracks the number of fruits that are damaged during the harvesting process, including damages to the fruit stems, relative to the total number of ripe fruits identified.

2.3 Innovative harvester robots: cutting-edge developments from startups and academia

In the following section, several of the most innovative autonomous robots designed for harvesting will be described. These robots, developed by both leading startups and academic institutions, highlight the cutting-edge developments driving the future of automated harvesting.

"An automated apple harvesting robot—From system design to field evaluation"

This work has been published in 2023 [8]. With the goal of designing a robot for selective apple harvesting, the proposed unified system includes a perception component, a low-cost 4 DOF manipulator, a vacuum-based soft gripper, and a collector to gather and transfer the picked fruits. By fusing the data recorded by the RGB-D camera and the laser-camera unit, a novel perception DL algorithm can precisely locate the target apple, achieving millimeter-level localization performance. The 4 DOF manipulator, with a simple design but high efficiency, approaches the target object under accurate motion planning and control. The soft end effector, that grasps the apple, is attached to the vacuum tube's inlet for gripping fruit: this design adapts easily to various fruit shapes and orientations, and helps prevent bruising during harvesting. Finally, the collector fetches the picked fruit. By evaluating the performances of the robot in two orchards, with two different tree architectures and leaves distribution, the robot registered respectively 82.4% and 65.2% of successful harvesting rate, with 6 s as average cycle time to pick a fruit.

"A multi-arm robot system for efficient apple harvesting: Perception, task plan and control"

This work dates back to 2023 [9]. This system integrates a multi-arm robotic design with a perception system, advanced task planning and control mechanisms. The system features a perception component that includes a multi-task deep learning algorithm utilizing stereo vision cameras for precise fruit recognition and localization despite occlusions and varying lighting conditions. The task planning

Figure 2.1: Representation of the harvester robot with the 4-DOF manipulator from [8]

is based on a Markov game framework using multi-agent reinforcement learning (MARL), which optimizes the harvesting sequence to reduce operational time and avoid inter-arm collisions. The robot, equipped with four 3-DOF arms and custom-designed grippers, is capable of efficient fruit picking in complex orchard environments. Field experiments demonstrated that the proposed system reduced fruit localization errors by 44.43% and decreased the average cycle time by 33.3% compared to heuristic-based methods. The robot obtained a harvesting success rate between 71.28% and 80.45%, with an average cycle time ranging from 5.8 s to 6.7 s, depending on the growth conditions of the apple trees.

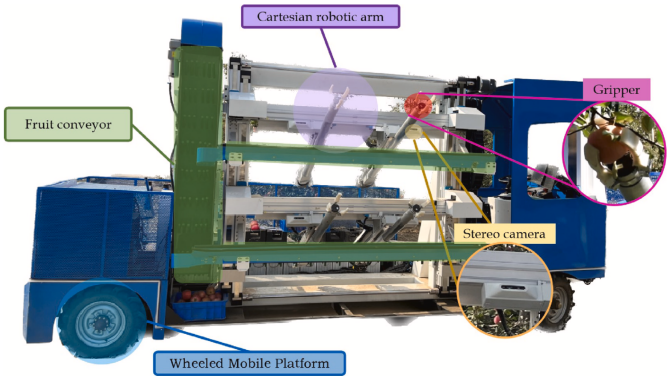


Figure 2.2: Representation of the multi-arm harvester robot's hardware from [9]

12 arms solution: FFRobots

The FFRobot developed by FFRobotics is an advanced, fully automated robot designed to address the task of fruit harvesting [10, 11]. This robot features a platform providing power, moving the machine and holding the fruit collectors; 12 independent robotic arms (six on each side of the basis, positioned at different heights), each equipped with cameras and grippers that can grasp, rotate, and cut fruit stems; a unified component to handle and fill the boxes. Guided by AI and sophisticated vision systems, and developed with ROS, the robot can identify, locate, and harvest fruits. After the gripper grasps the fruit, each robotic arm withdraws to the basis of the FFRobot and gently places the fruit onto a conveyor belt. This belt transports the fruit to a bin located at the bottom of the platform. The platform, holding three bins, automatically lowers a full bin to the ground and replaces it with an empty one to continue the harvesting process. After harvesting the fruits from trees on both sides, the platform moves to the next section and repeats the process. It continues this cycle until it reaches the end of the row, at which point it turns and begins harvesting the next row. In a nutshell, the FFRobot gains a rate of up to 9000 fruits per hour, it can work 20 hours per day, it typically collects around 90% of the fruit from a tree in an orchard arranged with fruiting walls, and it causes minimal damage to the fruit, with only about 5% affected.

Figure 2.3: Representation of the 12 arms FFRobot from [10]

Flying autonomous harvesters: Tevel

Tevel, developed by Tevel Aerobotics Technologies, represents an innovative system, specifically tailored for fruit harvesting [12]. This system incorporates flying robotic drones equipped with high-resolution cameras and sophisticated sensors, featuring lightweight and aerodynamic design, that enables them to hover and maneuver precisely and efficiently among the branches of fruit trees. As end-effector, the drones use a suction cup to pick the fruits, placed at the end of their arm. The robotic drones are powered by: maneuver algorithms (to optimize the trajectory planning), balancing algorithms (to counterbalance the forces applied by the greenery and fruits), AI perception algorithms (for fruit tracking and data fusion), vision algorithms (to detect the target fruits and the other objects and to classify fruits according to size and ripeness), harvesting optimization (to manage the fleet according to gathered orchard data). The robots appear versatile, able of multi-tasking and handling various fruit types (apples, pears, peaches, apricots, nectarines, plums, avocados), orchard designs and agricultural platforms. The system provides real-time information regarding fruit quantity, weight, color grading, ripeness, diameter, timestamp, geolocation, and it's able to harvest 24/7. What's more, the system is cost-effective and environmentally friendly, reducing food waste.

Figure 2.4: Representation of the flying harvester Tevel from [12]

Chapter 3

Theoretical Background

This chapter aims to introduce the theoretical foundations on which the designed architecture comprising visual perception and control for apple harvesting is based. In the first section, an overview of various methods and techniques used for visual servoing is presented, with a particular attention on Image-Based Visual Servoing (IBVS). The second section shifts the focus to the computer vision tasks essential for the system's operation, the detection and segmentation of the target object, and the deep learning architecture (CNN) through which the tasks are accomplished.

3.1 Introduction to Visual Servoing

Visual servoing, also known as vision-based robot control, is a technique that uses visual information to control the movements of a robotic system [13]. One or more cameras capture the scene from a starting pose, resulting in the current image. The desired image is represented by the scene as seen from the desired camera pose. The basic problem lies in finding a camera motion that guarantees to move from the initial pose to the desired one, exploiting the time-varying image as input. Visual Servoing (VS) integrates concepts and methods from several research areas, including image processing and computer vision to extract visual data and features from camera, artificial intelligence and machine learning for advanced image processing and decision-making capabilities, and robotics and control theory to develop control algorithms for robotic manipulators that continuously update the robot's position and orientation based on visual feedback.
Due to its precision and adaptive interaction with the environment, VS has a wide range of applications across various fields [14], including:

- Medical surgery, assisting surgeons with precise movements during minimally invasive procedures

- Autonomous navigation of unmanned aerial vehicles or driverless cars, improving the safety and reliability of the systems by enabling them to navigate and interact with their surroundings
- Agriculture, automating tasks like harvesting, planting and monitoring crops to increase efficiency and yield
- Industrial Automation, enhancing the flexibility and accuracy of robots in manufacturing processes such as assembly
- Service robotics, improving the performance of robots in domestic and commercial settings, such as cleaning, security and customer service.

By leveraging visual information, visual servoing empowers robots to perform complex and delicate tasks with a high degree of autonomy and precision, making it a vital technology in advancing the capabilities of modern robotics.

3.1.1 Classification of Visual Servoing systems: configurations and approaches

Researchers have been exploring visual servoing for more than four decades. But it was thanks to the advent of high speed processors and cameras that VS systems, implemented in real-time contexts, have found applications in industry. According to differences in the configurations and implementations, a sort of classification for VS systems is proposed in the following [14]:

- Camera configuration: Eye-in-hand vs. Eye-to-hand
- Number of cameras: Mono Vision, Stereo Vision, Multiple Cameras
- Control scheme: IBVS, PBVS or Hybrid
- VS controller: Proportional, Adaptive or Model Predictive Controller
- Image features: Point features, Line features, Image Moments features

3.1.2 Camera setup

Eye-in-hand vs. Eye-to-hand

The first classification of visual servoing systems is based on camera positioning. Eye-in-hand configuration consists in mounting the camera on the robot's end effector. Advantages include the high precision due to the direct and consistent view of the manipulation area that reduces the risk of occlusions and ensures continuous

tracking, improved 3D perception. However, disadvantages are limited field of view, motion-induced artifacts (such as motion blur and perspective changes) and frequent, complex calibration.
Eye-to-hand configuration fixes the camera in the environment. This setup provides several pros, including a stable viewpoint, a larger field of view covering the entire workspace and easier calibration. Its downsides include frequent occlusions from the robot or other objects, reduced precision for small movements if the camera is far from the task and complexity in 3D perception, often requiring multiple cameras or sophisticated algorithms.
The choice between configurations depends on task requirements. Eye-to-hand is preferable for broad workspace views, multi-robot coordination, or simpler calibration processes. Eye-in-hand suits high-precision tasks, continuous end effector tracking and detailed 3D perception [14].

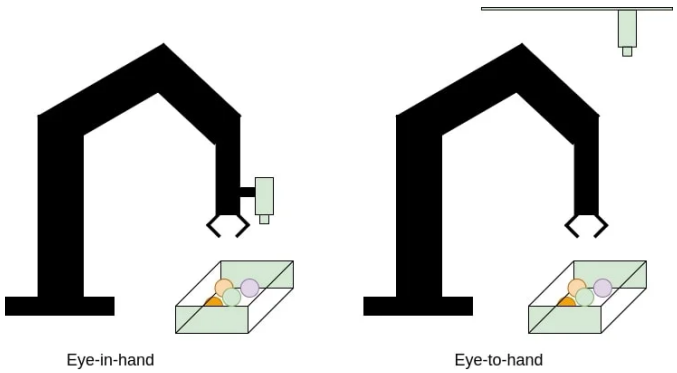


Figure 3.1: Camera configurations for visual servoing from [15]

Number of Cameras

As second way of classifying VS systems is by the number of cameras: Monocular cameras, Stereo Vision or Multiple cameras, each compatible with eye-to-hand or eye-in-hand configurations [14, 16].
Monocular camera systems (Single Camera) capture 2D images, providing information about the image plane but lacking direct depth perception, requiring techniques to infer depth. They have a restricted view but faster processing. Depth-Enhanced Monocular Cameras also include depth sensors, making them ideal for mobile robots and drones.
Stereo vision systems employ two or more cameras at known distances, calculating depth via triangulation. They're rarely used in eye-in-hand configurations due to narrower field of view.
Multi-Camera Systems utilize more than two cameras to cover a larger area or

capture different perspectives of the same scene, improving robustness and accuracy. Even so, multiple cameras lead to more data to process, requiring efficient algorithms and powerful hardware for processing [14, 16].

3.1.3 Visual servoing controllers

Proportional, Adaptive or Model Predictive Controller for VS systems

The fundamental controller used across all three visual strategies (IBVS, PBVS, Hybird) is the proportional controller, which reduces feature errors exponentially. However, more sophisticated controllers have been developed to address its limitations. Adaptive control estimates unknown or uncertain system parameters, such as camera calibration or object depth. Robust visual servoing enhances system stability in the presence of significant calibration errors. Model predictive controllers cope with system constraints like image boundaries and joint limitations during robot movement. Non-linear controllers, such as sliding mode control, improve robustness in feature trajectory tracking [14].

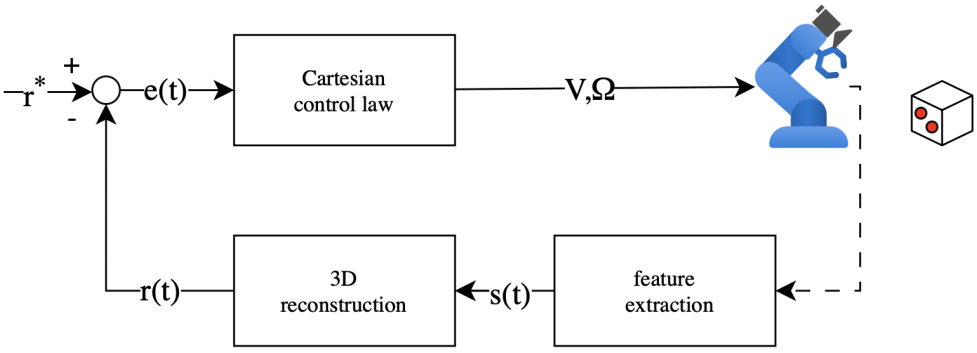
Image Features

An image feature is an interesting characteristic or geometric structure extracted from an image or scene. Some examples could be: points, lines, ellipses (or other 2D contour). The feature parameter is instead any numerical quantity associated to the chosen feature in the image plane, such as: the coordinates of a point, line's angular coefficient and offset, circle's center and radius, 2D contour's area and center, object's generalized moments in the image. Using moments avoids the problem of finding correspondences between points but complicates the control of all six degrees of freedom of the camera when when used as the sole feature [17].

Control schemes

A further classification of the VS schemes is made on the basis of how the error is generated. The three most famous architectures are: Position-Based Visual Servo, Image-Based Visual Servo and Hybrid Visual Servo. As we can see in Fig. 3.2, in Position-Based Visual Servoing (PBVS), image features are utilized to determine the current 3D pose of an object. This current pose is then compared with a predefined desired 3D pose to produce a Cartesian pose error signal, which guides the robot towards the target position. Clearly, to compute the object pose, this strategy needs an accurate camera calibration (to map the 2D data of the image features to the Cartesian space data) and the geometric model of the object. The sensitivity to calibration errors and the 3D pose reconstruction's computational cost are the main drawbacks of the PBVS. Furthermore, since there is no control on the image

trajectory, the PBVS fails when the object looses the camera's field of view. On



r*: desired pose
r(t): reconstructed current pose
V,Ω: Cartesian commands of the kinematic type
s(t): current visual features

Figure 3.2: PBVS control scheme

the contrary, as we can see in Fig. 3.3, in the IBVS strategy, the error is calculated directly in the image space: the desired features are compared with the current features of the object. The motion of the robot aims at overlapping the current features (thus, what it sees from the camera) with the desired ones. Thus, it's almost insensitive to intrinsic/extrinsic camera calibration parameters, it does not need any geometric model of the object and it presents a lower computational cost compared with PBVS. The uncontrolled path in 3D space can make the robot move beyond its allowed joint angles, especially when big turns and moves are needed to reach the goal. Additionally, this approach is hampered by potential singularities and possibility of encountering local minima. The hybrid VS scheme combines the two earlier approaches. It controls the robot by handling the end-effector's rotational movement and translational movement separately. Hybrid VS mixes the best of image-based and position-based methods to enhance control accuracy and robustness. In this way, the system can effectively manage complex tasks where both fine rotational adjustments and large translational movements are necessary, while mitigating issues such as singularities and joint limit violations [14].
Due to its numerous advantages, such as robustness and lower computational cost, the IBVS (Image-Based Visual Servoing) architecture has been preferred for this thesis work. Therefore, the following section will provide a detailed analysis of the steps involved in designing an IBVS controller.

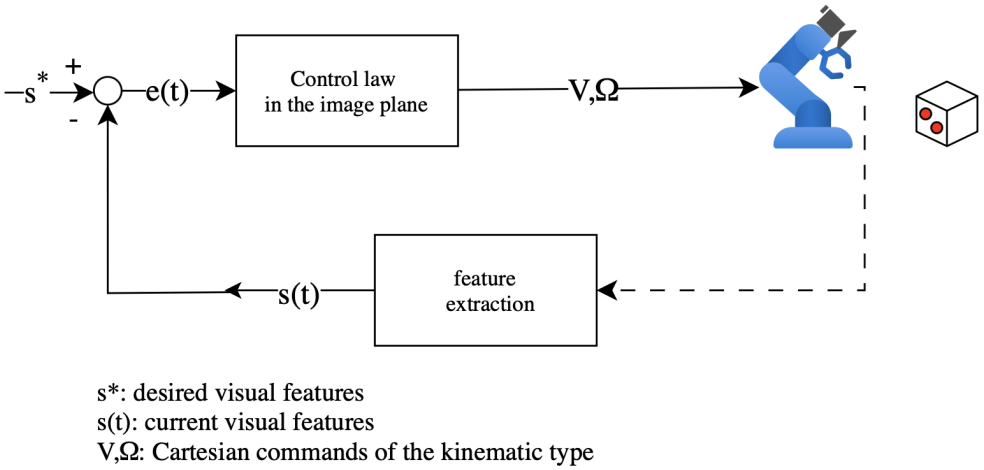


Figure 3.3: IBVS control scheme

3.1.4 IBVS Controller Design

The goal of IBVS consists of minimizing the following error, defined in the image feature space:

$$\mathbf{e}(t) = \mathbf{s}(\mathbf{m}(t), \mathbf{a}) - \mathbf{s}^* \tag{3.1}$$

where:

- $\mathbf{m}(t)$: collection of image measurements (the aformentioned image features parameters);
- $\mathbf{s}(\mathbf{m}(t), \mathbf{a})$: vector of k visual features, extracted directly from vision data (e.g., centroid of a detected object, the vertices of the bounding box of the detected object, and so on);
- $\mathbf{a}$: set of parameters providing additional information about the system (camera intrinsic parameters in this scheme, to go from image measurements expressed in pixels to the features);
- $\mathbf{s}^*$: vector of desired values of the features.

Basic assumptions

As previously explained, since a wide variety of control schemes are available, it is necessary to make certain assumptions about the scheme used in our thesis work:

- **Eye-in-hand systems**: The camera is attached to a six degree-of-freedom robot's end effector and it is treated as a free-moving object.

- **Static targets**: The targets are assumed to be motionless, thus $\mathbf{s}^*$ is constant over time and $\mathbf{s}$ varies according to the camera motion.
- **Purely kinematic systems**: The dynamics of camera motion are neglected; it is assumed that the camera can precisely carry out the applied velocity control.
- **Perspective projection**: The imaging geometry is modeled as a pinhole camera.

We can then proceed with the discussion by delving into the design of the control law.

Design of the control law

Given a set of features parameters $\mathbf{s} = \begin{bmatrix} s_1 & \cdots & s_k \end{bmatrix}^T \in \mathbb{R}^k$, the equation 3.2 expresses the kinematic differential relationship between the time variation of $\mathbf{s}$ (the motion of features) and the movement imposed to the camera.

$$\dot{\mathbf{s}} = \mathbf{L}_s \mathbf{v}_c = \mathbf{L}_s \begin{bmatrix} \mathbf{v}_c \\ \omega_c \end{bmatrix} \tag{3.2}$$

Here, $\mathbf{v}_c = (v_c, \omega_c) \in \mathbb{R}^6$ is the spatial velocity of the camera expressed in the camera frame $\mathcal{F}_c$, being v_c the instantaneous linear velocity of the origin of the camera frame and ω_c the instantaneous angular velocity of the camera frame; $\mathbf{L}_s$, $k \times 6$ matrix, is named *interaction matrix*, also known as *feature Jacobian* related to $\mathbf{s}$.
By using equations 3.1 and 3.2, we can directly derive the relationship between the camera velocity and the temporal variation of the error:

$$\dot{\mathbf{e}} = \mathbf{L}_e \mathbf{v}_c \tag{3.3}$$

with $\mathbf{L}_e = \mathbf{L}_s$. If we look at $\mathbf{v}_c$ as the input to the robot controller and aim to achieve an exponentially decoupled reduction of the error (that is, $\dot{\mathbf{e}} = -\lambda \mathbf{e}$) , we derive the following from equation 3.3:

$$\mathbf{v_c} = -\lambda \mathbf{L}_e^+ \mathbf{e} \tag{3.4}$$

with $\mathbf{L}_e^+ \in \mathbb{R}^{6 \times k}$, being $\mathbf{L}_e^+ = (\mathbf{L}_e^T \mathbf{L}_e)^{-1} \mathbf{L}_e^T$, the Moore-Penrose pseudoinverse of $\mathbf{L}_e$ if $\mathbf{L}_e$ is of full rank 6. This approach minimizes both $\|\dot{\mathbf{e}} - \lambda \mathbf{L}_e \mathbf{L}_e^+ \mathbf{e}\|$ and $\|\mathbf{v}_c\|$. When $k = 6$ and $\det \mathbf{L} \neq 0$, it is possible to invert $\mathbf{L}_e$, resulting in the control law $\mathbf{v_c} = -\lambda \mathbf{L}_e^{-1} \mathbf{e}$ [13].
In practice, it appears impossible to know exactly either $\mathbf{L}_e$ or $\mathbf{L}_e^+$, leading to an

approximation or estimation of the Jacobian matrix, denoted by $\widehat{\mathbf{L}_e^+}$.
The control law can be rewritten as [13]:

$$\mathbf{v_c} = -\lambda \widehat{\mathbf{L}_e^+} \mathbf{e} \tag{3.5}$$

After having obtained the control law for Image Based Visual controllers, we'll continue the discussion by analyzing the structure of the Jacobian matrix. To do this, a brief mention to the pinhole model is following.

Pinhole camera model

The pinhole camera model, originating from the camera obscura described by Leonardo da Vinci in 1502, represents the simplest camera model. The camera is conceptualized as a pinhole camera where light passes through a small aperture and projects an inverted image onto a projection plane. Let's now focus on the geometry of this projection. Let's consider a 3D coordinate system with the origin at the camera's pinhole (camera frame). A 3D point with coordinates $\mathbf{P} = (X, Y, Z)$ is mapped onto a 2D image plane with coordinates $\mathbf{p} = (x, y)$. The image plane is located at a focal distance f from the pinhole. By considering similar triangles, the projection of the point (X, Y, Z) onto the image plane is given by:

$$x = \frac{fX}{Z}, \quad y = \frac{fY}{Z} \tag{3.6}$$

These equations describe the ideal projection, neglecting geometric distortions or blurring of unfocused objects. Let's go on with the representation of the image plane points in homogeneous coordinates: $\tilde{\mathbf{p}} = (x', y', z')^T$. The tilde shows that the vector is being written in homogeneous coordinates.

$$\tilde{\mathbf{p}} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.7}$$

Until now, we have considered the coordinates of the point in the camera reference frame. The need arises, for example, when there are multiple cameras in a system, to transform the coordinates into a different reference frame: the world or the robot RF. If we denote the coordinates in the camera RF with the apex ^{C} and the coordinates in the world RF with the apex ^{0}, we can explicitly state the relationship between the two reference frames:

$$\tilde{\mathbf{P}}^0 = \mathbf{T}_C^0 \cdot \tilde{\mathbf{P}}^C \tag{3.8}$$

where the matrix $\mathbf{T}_C^0$ is a transformation matrix that converts coordinates from the camera RF to the world RF.

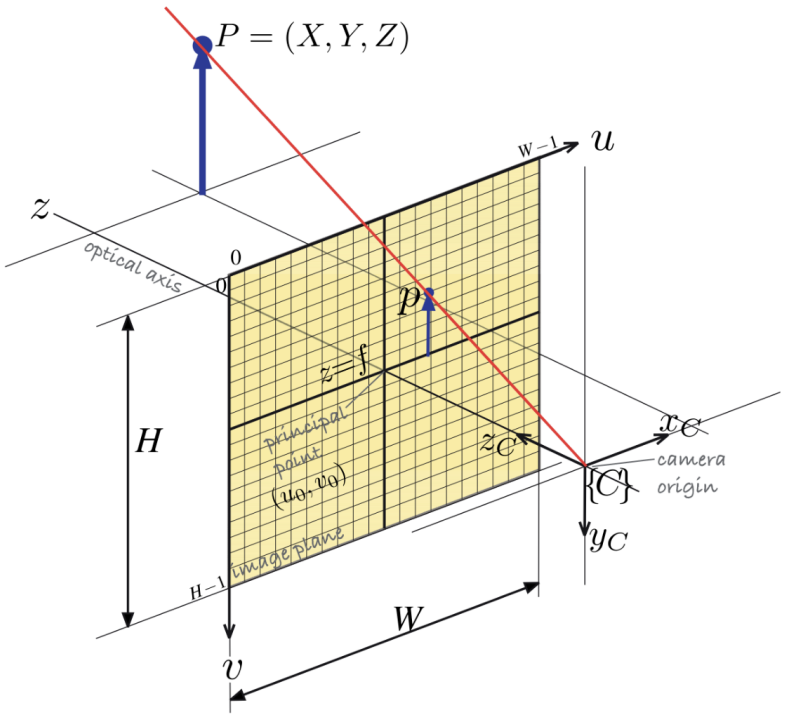


Figure 3.4: Central perspective imaging model from [18]

In digital cameras, the image plane is not continuous but organized as a grid of pixels or photodiodes. These pixels are identified by a two-dimensional vector (u, v) representing the horizontal and vertical indices of the grid. The pixel at coordinates (0,0) is located at the top-left corner instead of the center as shown in Fig. 3.4. The transformation between pixel coordinates (u, v) and image plane coordinates (x, y) is done as follows:

$$u = \frac{x}{\rho_w} + u_0, \qquad v = \frac{y}{\rho_h} + v_0 \tag{3.9}$$

where ρ_w is the width of a pixel in the image plane units, ρ_h is the height of a pixel in the image plane units; (u_0, v_0) are the coordinates of the principal point (obtained by intersecting the image plane with the optical axis, the z axis of the camera RF). By merging Eq. 3.6 and 3.9, we get:

$$\begin{aligned} u &= \frac{fX}{\rho_w Z} + u_0 = f_w \frac{X}{Z} + u_0 \\ v &= \frac{fY}{\rho_h Z} + v_0 = f_h \frac{Y}{Z} + v_0 \end{aligned} \tag{3.10}$$

It can be written in homogeneous coordinates as well, highlighting that f_w and f_h are adimensional quantities, obtained by the division of two distances $[mm]$:

$$\tilde{\mathbf{p}} = \begin{pmatrix} u' \\ v' \\ w' \end{pmatrix} = \begin{pmatrix} f_w \cdot X + u_0 \cdot Z \\ f_h \cdot Y + v_0 \cdot Z \\ Z \end{pmatrix} \tag{3.11}$$

In order to go from the homogeneous pixel coordinates to the non-homogeneous ones:

$$u = \frac{u'}{w'}, \quad v = \frac{v'}{w'} \tag{3.12}$$

Now, let's rewrite the perspective projection transformation in homogeneous coordinates in a linear way:

$$\tilde{\mathbf{p}} = \begin{pmatrix} u' \\ v' \\ w' \end{pmatrix} = \begin{bmatrix} f_w & 0 & u_0 \\ 0 & f_h & v_0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot (\mathbf{T}_C^0)^{-1} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \mathbf{KN}(\mathbf{T}_C^0)^{-1}\tilde{\mathbf{P}}^0 = \mathbf{C}\tilde{\mathbf{P}}^0 \tag{3.13}$$

where:

- $\mathbf{K}$ represents the camera parameters matrix;
- $\mathbf{N}$ is termed the projection matrix;
- $\mathbf{KN}$ is the intrinsic matrix, establishing the relation between pixel coordinates in homogeneous form and the spatial coordinates in the camera RF;
- $(\mathbf{T}_C^0)^{-1}$ constitutes the extrinsic matrix;
- $\mathbf{C} = \mathbf{KN}(\mathbf{T}_C^0)^{-1}$ is a 3×4 homogeneous transformation that maps a point $\tilde{\mathbf{P}}^0$ in homogeneous world coordinates to its corresponding point $\mathbf{x}$ in homogeneous image RF coordinates.

To conclude, employing the non-linear Eq. 3.12, one can derive the pixel coordinates (u, v) that correspond to an arbitrary point $\mathbf{P}^0$ in world space.

Interaction matrix

Our goal is deriving now the Interaction matrix. As we have stated, the Jacobian matrix is always related to a particular feature parameter. Since in this thesis work the coordinates of precise points will be choosen as image feature parameters, let's compute the interaction matrix for a single point with coordinates (X, Y, Z).

Firstly, the computation of time derivatives is needed for x, y. By applying the quotient rule to Eq. 3.6 [19], we obtain:

$$\dot{x} = f\frac{Z\dot{X} - X\dot{Z}}{Z^2}, \quad \dot{y} = f\frac{Z\dot{Y} - Y\dot{Z}}{Z^2} \tag{3.14}$$

By rewriting Eq. 3.6 as $X = \frac{xZ}{f}$ and $Y = \frac{yZ}{f}$, we can substitute these into $\dot{x}$ and $\dot{y}$, getting:

$$\dot{x} = \frac{f\dot{X}}{Z} - \frac{x\dot{Z}}{Z}, \quad \dot{y} = \frac{f\dot{Y}}{Z} - \frac{y\dot{Z}}{Z} \tag{3.15}$$

We can now express $\dot{X}$, $\dot{Y}$, and $\dot{Z}$ in terms of the camera velocity screw $\mathbf{v}_c$ and X, Y, Z. By writing the velocity of the fixed point P with respect to the camera frame [19], we get the equations of $\dot{X}, \dot{Y}$ and $\dot{Z}$:

$$\begin{gathered}
\dot{P} = -\mathbf{v} - \boldsymbol{\omega} \times P \\
\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = -\mathbf{v} - \boldsymbol{\omega} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \\
\begin{aligned}
\dot{X} &= -v_x - w_y Z + w_z Y \\
\dot{Y} &= -v_y - w_z X + w_x Z \\
\dot{Z} &= -v_z - w_x Y + w_y X
\end{aligned}
\end{gathered} \tag{3.16}$$

Going on blending the equations and playing with algebra, we obtain:

$$\begin{aligned}
\dot{x} &= -\frac{f}{z}v_x + \frac{x}{z}v_z + \frac{xy}{f}\omega_x - \frac{(f^2 + x^2)}{f}\omega_y + y\omega_z \\
\dot{y} &= -\frac{f}{z}v_y + \frac{y}{z}v_z + \frac{(f^2 + y^2)}{f}\omega_x - \frac{xy}{f}\omega_y - x\omega_z
\end{aligned} \tag{3.17}$$

Finally, the matrix form of the equations is given by [19]:

$$\mathbf{L} = \begin{bmatrix} -\frac{f}{Z} & 0 & \frac{x}{Z} & \frac{xy}{f} & -\frac{f^2+x^2}{f} & y \\ 0 & -\frac{f}{Z} & \frac{y}{Z} & \frac{f^2+y^2}{f} & -\frac{xy}{f} & -x \end{bmatrix} \tag{3.18}$$

As a compact form, we can write:

$$\dot{\mathbf{s}} = \mathbf{L}(s, Z)\mathbf{v_c} \tag{3.19}$$

It may be interesting to look at the nullspace of the Interaction or Jacobian matrix [19]:

$$\begin{bmatrix} x \\ y \\ f \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ x \\ y \\ f \end{bmatrix} \begin{bmatrix} xyZ \\ -(x^2+f^2)Z \\ fyZ \\ -f^2 \\ 0 \\ xf \end{bmatrix} \begin{bmatrix} f(x^2+y^2+f^2)Z \\ 0 \\ -x(x^2+y^2+f^2)Z \\ xyf \\ -(x^2+f^2)Z \\ xf^2 \end{bmatrix} \tag{3.20}$$

By intuition, it corresponds to all point movements that cannot be detected by the camera, for example: the translation and rotation about the projection ray, the translation and rotation about the y-axis of the camera RF, while maintaining proper orientation through respectively linear and rotational adjustments [19].
As a result, it's not feasible to control all 6 DOF for the camera movement using just one point in the image. A practical approach to address this limitation is to include more than one image point. It follows that the overall interaction matrix is obtained by stacking all single point interaction matrices (where the knowledge or estimate of the depth Z_i is needed for each point) [19, 13].

$$\dot{\mathbf{s}} = \begin{bmatrix} \dot{s}_1(t) \\ ... \\ \dot{s}_n(t) \end{bmatrix} = \begin{bmatrix} L_1(s_1, Z_1) \\ ... \\ L_n(s_n, Z_n) \end{bmatrix} \mathbf{v}_c \tag{3.21}$$

By adopting this approach, it is required to have at least three image points ($k > 6$) to control the 6 DOF. Nevertheless, there are certain setups where $\mathbf{L}$ matrix becomes singular. Additionally, there are four distinct camera poses where the error e equals zero, indicating the presence of four global minima that are indistinguishable from one another. Consequently, it is common to use more than three points in these scenarios [13].

Approximation of the Interaction Matrix

There are several methods for constructing the estimate $\widehat{\mathbf{L}_e^+}$ used in the control law [13, 19]:

1. $\widehat{\mathbf{L}_e^+} = \mathbf{L}_e^+$: in this construction we use the current depth Z_i of each point, without any approximation. However, in practical situations, these parameters must be estimated during each iteration according to pose estimation techniques, with the risk of an intensive computational load.

2. $\widehat{\mathbf{L}_e^+} = \mathbf{L}_{e^*}^+$: where the matrix $L_{e^*}^+$ corresponds to the value of L_e^+ at the desired position, $e = e^* = 0$. L_e is kept constant, only the desired depth for each point must be set, meaning that there is no 3D evolution overall the visual servoing process and the computational load is reduced.

3. $\widehat{\mathbf{L}_e^+} = 1/2(\mathbf{L}_e + \mathbf{L}_{e^*})^+$: proposed in [20], as a combination of the previous two, it requires both the current depth for each point and the desired one. This approach aims to balance the advantages of the two aforementioned methods by providing a compromise between accuracy and computational complexity.

The behavior of control schemes can be understood geometrically. For instance, when using $\mathbf{L}_e^+$, the control scheme ensures an exponential decrease of the error e. The image point trajectories ideally follow straight lines from their initial to desired positions. Some failures can occur because of incorrect interaction matrix's feature choices and column coupling, especially when the rotation between initial and desired configurations is large. This can lead to a retreating translational motion along the optical axis and inefficient control for rotations near π radians. To sum it up, if the error is small, the behavior is locally satisfactory.
Using $\mathbf{L}_{e^*}^+$ results in image motion similar to the previous method and sometimes it may attenuate the unwanted retreating motion.
$\widehat{\mathbf{L}_e^+} = 1/2(\mathbf{L}_e + \mathbf{L}_{e^*})^+$ matrix is the mean of the two matrices. Generally speaking, the resulting image motion remains consistent, even with large errors [13].

Stability Analysis of IBVS

In order to evaluate the stability of closed-loop VS systems, Lyapunov analysis is adoperated. As candidate Lyapunov function, the squared error norm $\mathcal{L} = \frac{1}{2}\|\mathbf{e}(t)\|^2$ is chosen. Let's compute its derivative: $\dot{\mathcal{L}} = \mathbf{e}^\top \dot{\mathbf{e}} = -\lambda \mathbf{e}^\top \mathbf{L}_e \widehat{\mathbf{L}_e^+} \mathbf{e}$. Then, the global asymptotic stability of the system is guaranteed if the sufficient condition holds [13]:

$$\mathbf{L}_e \widehat{\mathbf{L}_e^+} > 0 \tag{3.22}$$

meaning that we have asymptotic stability when the matrix $\mathbf{L}_e \widehat{\mathbf{L}_e^+}$ is positive definite.
For IBVS, which often has more features than the camera's degrees of freedom ($k > 6$), global asymptotic stability cannot be guaranteed. This is because the product $\mathbf{L}_e \widehat{\mathbf{L}_e^+}$ may have a nontrivial null space, leading to local minima where the error does not reduce to zero, despite an exponential decrease in each error component. Consequently, IBVS can only achieve local asymptotic stability, where the system may converge to a local minimum rather than the desired configuration. For further investigation and demonstration of stability analysis of IBVS, we invite the reader to look at [13].

3.2 Introduction to Deep Learning methods for Computer Vision tasks

Computer vision encompasses several crucial tasks that enable machines to interpret and analyze visual information. Two fundamental tasks in this domain include Object Detection (OD) and Instance Segmentation (IS).
Object detection involves identifying and localizing objects within an image, typically outputting bounding boxes and class labels for each detected object.
Instance Segmentation goes a step further identifying objects within an image but also distinguishing individual objects at the pixel level. Unlike semantic segmentation, which assigns a class label to every pixel without distinguishing between different instances of the same class, instance segmentation generates pixel-level masks, as well as providing the bounding box and class label, for each instance.
The described computer vision tasks can be addressed through various methods,

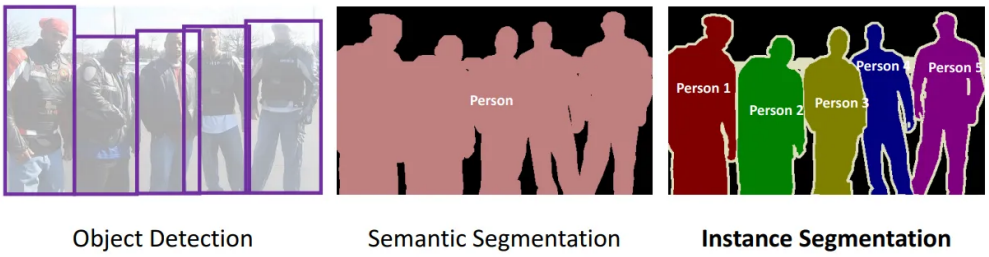


Figure 3.5: Examples of Object detection, Semantic Segmentation and Instance Segmentation from [21]

with deep learning techniques, particularly Convolutional Neural Networks (CNNs), emerging as the most effective. Frameworks like YOLO (You Only Look Once) have revolutionized these fields by offering real-time performance without sacrificing accuracy. The following sections will delve into the fundamentals of deep learning, with a specific focus on CNNs and the YOLOv8 architecture, which has been employed in this thesis for Object Detection and Instance Segmentation tasks [22].

3.2.1 Artificial Intelligence

It is common to get confused about the concepts related to AI, ML, DL. In this paragraph, we want to clarify each of these notions.
Artificial intelligence (AI) refers to all technologies allowing computers and machines to mimic the human intelligence and problem-solving [23]. As shown in Fig. 3.6, Machine Learning and Deep Learning are two sub-fields of AI.
In particular, Machine Learning is the scientific discipline focused on developing

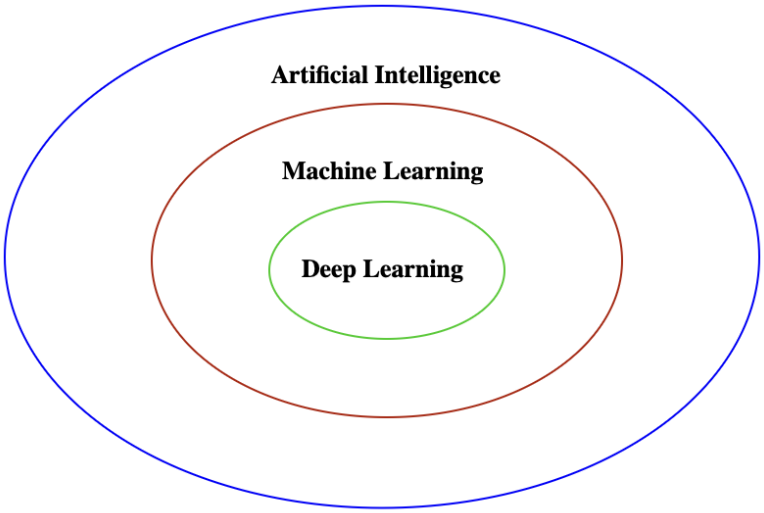


Figure 3.6: Artificial Intelligence and its subcategories

algorithms and statistical models that enable computers to execute specific tasks by learning from data rather than following explicit instructions.
As a subset of ML, Deep Learning teaches computer to process data and create patterns to use in decision making, by adopting learning models referred as deep neural networks, which are inspired by the human brain.
The key difference between the two branches lies in the fact that:

- in **ML**: the feature extraction is performed by a human operator;
- in **DL**: the feature extraction is accomplished by the *deep* learning network itself.

3.2.2 Neural networks

Before diving into CNNs, a brief introduction to Neural Networks is needed. The origin of these networks dates back to Frank Rosenblatt's work on perceptrons [24]. Neural Networks are nets of interconnected nodes or artificial neurons in an acyclic graph, organized in a layered structure, resembling the human brain. The output of the neurons in one layer becomes the input to the neurons of the next layer, without forming cycles that loop back to earlier neurons. As already stated, these neurons are usually arranged in layers, including an input layer, one or more hidden layers and an output layer. These networks are called Multilinear Perceptron (MLP) or Artificial Neural Network (ANN). An N-layer network excludes the input layer when counting layers. A neural network is called *deep* when it has more than one hidden layer; otherwise, it is called *shallow*.

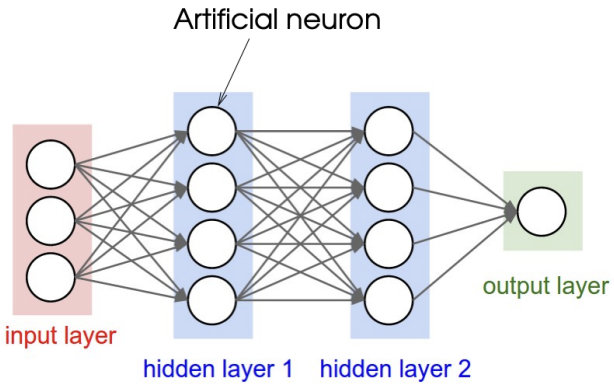


Figure 3.7: Representation of a 3-layer neural network with three inputs, two hidden layers of respectively 4 and 4 neurons, and one output layer from [25]

Let's now delve deeper into the nodes of the artificial networks: the neurons. The concept of the artificial neuron has evolved significantly over time. Initially, artificial neurons were mathematical functions designed to imitate the behavior of biological neurons. The early model, known as the McCulloch-Pitts neuron, used a weighted sum of inputs followed by a step function to determine the output. This basic model evolved into the perceptron, introduced by Frank Rosenblatt in 1958. The perceptron included a threshold logic unit (TLU) which used a linear combination of inputs and a threshold function to produce binary outputs, enabling it to classify data into two distinct categories. As neural network research progressed, the perceptron was extended into Multilayer Networks with Fully Connected Layers [26]. Mathematically, the neuron applies a nonlinearity (the well-known *activation*

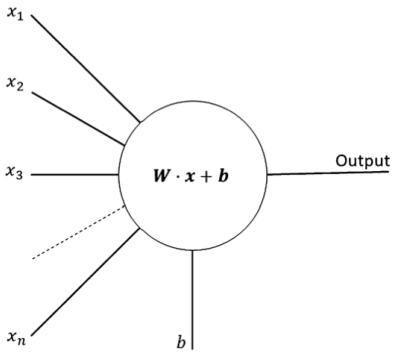


Figure 3.8: The schematic of a neuron from [24]

function) to an affine transformation. The input features $\mathbf{x} = (x_1, x_2, ..., x_n)$ are

processed through an affine function combined with a non-linearity φ:

$$T(x) = \varphi(\sum_i W_i x_i + b) = \varphi(\mathbf{W} \cdot \mathbf{x} + \mathbf{b}) \tag{3.23}$$

Here, $\mathbf{W}$ are given *weights* and $\mathbf{b}$ is a given *bias*.
The output of the weighted sum could theoretically be any value in the range $[-\infty, +\infty]$, but we aim at constraining the neuron's output within a specific range, such as [0,1], in a way that the neuron could be activated (fired) when the output is greater than a given threshold. That's the reason why the activation function is inserted. Various options are available for introducing the non-linearity: they will be accurately shown in the section about CNN.
Unlike the hidden layers, the output layer typically doesn't use a non-linearity [24].

The learning process of NNs

In NNs, the weights and biases are known as *learnable* parameters. Learning in this context refers to the process of finding the optimal values for these parameters. This is achieved through a process called training. In 1986, after failing attempts in finding a proper way to train MLPs, David Rumelhart, Geoffrey Hinton and Ronald Williams introduced the *backpropagation* training algorithm [26].

Loss function

Basically, it's necessary to introduce a measure to evaluate how the network is tweaking the value of weights and biases, thus how well the neural network's predictions match the actual target values. We can employ a *cost* function and the goal of training is minimizing this cost function. For a set of training examples, a popular cost function $C(w, b)$ is the Mean Squared Error (MSE).

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{3.24}$$

where:

- w and b represent the parameters (weights and biases) of the network;
- n is the total number of training examples x;
- a is the actual output of the network;
- $y(x)$ is the desired output.

The cost function quantifies the discrepancy between the network's actual output and the target output for a given input. Since the cost function depends on the

values of $\mathbf{W}$ and $\mathbf{b}$, if it approaches 0 for all training examples, it means that the learning process is effective and appropriate parameter values have been identified. To this aim, the minimization problem is solved by the Gradient Descent algorithm. If we denote with v a generic n-dimensional input array, for small variations of each variable v_j, the variation of the cost function is given by:

$$\nabla C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 + \ldots + \frac{\partial C}{\partial v_j}\Delta v_j + \ldots + \frac{\partial C}{\partial v_n}\Delta v_n \tag{3.25}$$

The variation ∇C in C produced by a small variation $\nabla v = (\nabla v_1, ..., \nabla v_n)^T$ is:

$$\nabla C \approx \nabla C \cdot \nabla v \tag{3.26}$$

with $\nabla C = (\frac{\partial C}{\partial v_1}, ..., \frac{\partial C}{\partial v_n})^T$. The algorithm works by iteratively adjusting the parameters in the direction of the steepest descent of the cost function (negative ∇C). Let's denote with α a parameter called the learning rate, a hyperparameter that controls the step size of each update:

$$\nabla v = v' - v = -\alpha \nabla C \tag{3.27}$$

By combining the equations 3.26 and 3.27, we obtain:

$$\nabla C \approx -\alpha \nabla C \cdot \nabla C = -\alpha \|\nabla C\|^2 \tag{3.28}$$

The update rule for the parameters in gradient descent is given by:

$$v' := v - \alpha \nabla C \tag{3.29}$$

We can extend this rule to the specific case of the weights and biases that we want to update during the epochs:

$$w_j' = w_j - \alpha \frac{\partial C}{\partial w_j}$$
$$b_j' = w_j - \alpha \frac{\partial C}{\partial b_j}$$

The concept of gradient descent can be improved by using the stochastic gradient descent (SGD) technique, which helps in faster convergence. While the traditional gradient descent involves computing the gradient of the cost function C over the entire training dataset, SGD accelerates this process by using only a small, randomly selected subset of the data, known as a mini-batch. This approximation allows for more frequent updates to the model parameters (weights w_j and biases b_j).
The update rules in SGD are modified as follows:

- For weights: $w'_j = w_j - \alpha \frac{\partial C}{m \partial w_j}$
- For biases: $b'_j = b_j - \alpha \frac{\partial C}{m \partial b_j}$

where m is the size of the mini-batch. The great advantage is that these updates are more frequent but less computationally expensive than in traditional gradient descent, leading to potentially faster convergence and better generalization when training neural networks [23].

Backpropagation algorithm

The Backpropagation algorithm is nothing more than a Gradient Descent for the automatic calculation of gradients. It's able to compute the gradient of the network's error with respect to each individual model parameter in two steps: one forward and one backward. It determines how to adjust each connection weight and bias term to minimize the error. Once these gradients are obtained, a standard Gradient Descent step is executed, and this process is repeated until the network converges to the optimal solution.
Let's see in details how the algorithm works. For a network architecture with L layers, let's suppose that all the weights and biases of the network have been initialized. A mini-batch of m inputs is randomly selected from the dataset and the following procedure implements a learning update using the gradient descent method on that subset [23]:

1. The training data are acquired.
2. For each training input x from the mini-batch:
   - it enters the input layer, following the feedforward pass, and it is sent to all following layers. For each layer $l = 2, ..., L$:

     $$z^{x,l} = w^l a^{x,l-1} + b^l$$

     and

     $$a^{x,l} = g^{[l]}(z^{x,l})$$

     are computed till the output, where $g^{[l]}$ is the activation function for layer l.
   - The network's output error is computed as:

     $$\delta^{x,L} = \nabla_a C_x \odot g'^{[L]}(z^{x,L})$$

     It consists of two terms: the first one expressing how fast the loss function is influenced by the output activations; the second term evaluates the variation of $g^{[l]}$ depending on the variation of the input $z^{x,L}$.

- The Backward Propagation is done by calculating:

$$\delta^{x,l} = ((w^{x,l+1})^T \delta^{x,l+1}) \odot g'^{[l]}(z^{x,l})$$

with $l = L-1, L-2, ...,2$. In this way, the algorithm measures how much of the error at each layer is attributed to the connections from the preceding layer, applying the chain rule repeatedly.

3. The weights and biases are updated with the SGD rules for each layer $l = L-1, L-2, ..,2$:

$$w^l = w^l - \frac{\alpha}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$$

$$b^l = b^l - \frac{\alpha}{m} \sum_x \delta^{x,l}$$

Naturally, to effectively implement stochastic gradient descent, an external loop that generates mini-batches of training examples is also needed, as well as another loop that iterates through multiple epochs of training.

Let's point out that this represents the generic working flow of the backpropagation algorithm but other versions are available. Further more, the number of epochs needed to complete the training process depends on the network's dimension and other factors affecting the process, such as the learning rate and the mini-batch size [23].

3.2.3 Convolutional Neural networks

Let's now introduce a new architecture of feed-forward neural networks: Convolutional Neural Networks (CNNs), also known as *convnets*, specifically designed to handle image-like inputs. Traditional fully connected networks can be inefficient in such cases for two reasons: they introduce a wide number of learnable parameters, slowing down the learning process, and they fail to account for the spatial structure inherent in images [24, 27].
Firstly, we'll explore the three key concepts about CNNs: local receptive fields, shared weights and pooling. Following that, we'll provide an overview of CNN architecture, composed by the following building blocks: Convolutional layers, Nonlinear activation functions, Pooling layers and Fully Connected layers.

Local Receptive Field

Inputs differ between traditional neural networks and CNNs. While neural networks process inputs as a flat line of data, CNNs handle inputs as a grid, such as a 28×28

array of neurons, with each value representing a pixel's intensity. Unlike fully connected layers, where each input pixel connects to all hidden neurons, CNNs associate each hidden neuron with only a small, localized region of the input image, known as a local receptive field [23]. Now, let's suppose we have a 5×5 local receptive field of input pixels. Each connection in this window learns a weight, and the hidden neuron also learns an overall bias. This hidden neuron essentially learns to interpret its specific local receptive field. In the forward pass, the window is then slid across the entire input image, with each new position activating a different hidden neuron in the first hidden layer. Even if illustrated this with a stride of one pixel, different stride lengths can also be adopted. Stride directly affects the spatial size of the convolution output. Using larger strides reduces the output's dimensions, while smaller strides preserve more spatial details. Although larger strides lower computational demand and speed up processing, they can also affect the quality.
In conclusion, the number of neurons in the hidden layer can be calculated as:

$$n_h = \frac{W - F - 2P}{S} + 1$$

where W represents the width of the image, F is the size of the receptive field, S denotes the stride (or the step size of the sliding window), P refers to the amount of zero-padding applied to the image. Zero-padding involves adding pixels with a value of zero along the image's borders, which helps in controlling the output size of the layer.

Shared weights

It is important to note that, in the aforementioned case of a local receptive field with 5×5 input pixels, each neuron in the hidden layer learns 5×5 weights and one bias. These 5×5 weights and the bias will be shared among all the hidden neurons, meaning that their values remain constant as the receptive field moves across the input image. That's the great advantage of CNNs: the learnable parameters are way reduced with respect to fully connected layers, implying faster trainings and the opportunity to build deeper networks. The output of the hidden neuron located at position j, k in the hidden layer can be expressed as:

$$out_{j,k} = \sigma(b + \sum_{l} \sum_{m} w_{l,m} a_{j+l,k+m})$$

where:

- b is the bias term
- $w_{l,m}$ are the weights corresponding to the receptive field

- $a_{j+l,k+m}$ are the input activations over the region of the input image that the receptive field covers

Intuitively, we can draw as a conclusion that each hidden layer is able to learn a feature, everywhere in the input image. Thus, CNNs are highly suited for capturing the translation invariance present in images. To sum it up, we refer to the mapping from the input layer to the hidden layer as a feature map. The shared weights and bias constitute a kernel or filter. Since one feature map is only capable of detecting a single kind of feature, in order to accomplish more complex tasks, we need more than one feature map: these features maps are stacked to get the output map or activation volume of the convolutional layer.
As a remark, the term convolution inherits the name from the mathematical operation. In this context, convolution refers to the process of applying a kernel (or filter) to the input image to extract features. Mathematically, the convolution operation involves sliding the kernel across the input image and computing the sum of element-wise multiplications between the kernel and the portion of the image it covers. This can be formally expressed as:

$$(I * K)(x, y) = \sum_i \sum_j I(x + i, y + j) \cdot K(i, j) \tag{3.30}$$

where I represents the input image, K is the kernel, and (x, y) indicates the position of the output feature map. The result of this operation is a feature map that highlights specific features detected by the kernel, such as edges or textures. This process enables CNNs to efficiently capture and represent spatial hierarchies and patterns within the image [23, 24].

Nonlinearities

As already mentioned in NNs and CNNs introduction, nonlinearities play a crucial role: they prevent the networks to barely compute a linear combination of the inputs. Furthermore, choosing the proper Nonlinear Activation can speed up the learning process. Let's delve into some of the most popular ones [24].

- **Sigmoid**: The activation function is expressed as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, x \in \mathbb{R}$$

  In this case, the output $\sigma(x) \in (0,1)$ for all $x \in \mathbb{R}$. Additionally, the sigmoid function σ is monotone increasing, with $\lim_{x\to\infty} \sigma(x) = 1$ and $\lim_{x\to-\infty} \sigma(x) = 0$. This property makes it well-suited for producing outputs in the range $[0, 1]$, such as probabilities or normalized images. However, it's important to note that as the input x is far from zero, the neuron saturates, causing the gradient

of $\sigma(x)$ with respect to x to approach zero. This hampers optimization, which is why sigmoid functions are seldom used in the intermediate layers of CNNs.

- **tanh**: The definition of the tanh is:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, x \in \mathbb{R}$$

The output $\tanh(x) \in (-1,1)$ for all $x \in \mathbb{R}$. Also in this case, *tanh* is monotone increasing as well and a risk of vanishing gradient is present: that's why *tanh* is not so used in intermediate layers of CNNs.

- **ReLU**: Expressed as:

$$\text{ReLU}(x) = \max(0, x), x \in \mathbb{R}$$

It is clear that the derivative of ReLU is $\text{ReLU}'(x) = 1$ when $x > 0$ and $\text{ReLU}'(x) = 0$ when $x < 0$. ReLU activation allows to speed up convergence, particularly when paired with a well-chosen weight initialization strategy and learning rate in CNNs.

- **softmax**: It is defined as:

$$\text{softmax}(x)_i := \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}, \quad x \in \mathbb{R}^n$$

It maps a vector $x \in \mathbb{R}^n$ into a probability vector of the same length n. By applying the transformation $x \to \exp(x)$, it preserves the order of elements, with subsequent normalization, ensuring the result is a valid probability distribution. This function is often used in classification tasks after the final fully connected layer in an n-class problem. However, softmax does not fully capture prediction uncertainty in cases with noisy labels.

Pooling layers

CNNs contain other particular layers called Pooling Layers. They immediately follow the convolutional layers. The goal of pooling is condensing and simplifying the information and reducing the spatial dimensions coming from the convolution. For example, each unit in the pooling layer might represent a region of 2×2 neurons from the preceding layer. There are two common methods for pooling: max-pooling and L2-pooling. The most common form of max-pooling uses a kernel size of 2×2 along with a stride of 2. This setup partitions the feature map into a regular grid of 2×2 blocks and then computes the maximum value within each block for every input feature. For L2-pooling, we calculate the square root of the sum of the squared activations within the 2×2 region [24, 23].

Overall architecture of CNN

Now that the main concepts regarding CNNs have been covered, we can take a look at how the entire architecture is built. A typical CNN architecture typically consists of several key components: convolutional layers, nonlinear activations, pooling layers, and finally, fully connected layers. The convolutional layers are responsible for detecting patterns in the input images, with each feature map capturing specific visual features. Following each convolution, a nonlinear activation function is applied to address the problem of vanishing gradients and to ensure that the network does not merely compute a linear combination of the inputs. Next, pooling layers are used to condense the most important features identified by the convolutional layers, reducing the dimensionality and helping to mitigate the risk of overfitting. Lastly, one or two fully connected layers are added on top of the CNN. The output of the CNN is flattened and transformed into a single vector, which is then processed by the fully connected layers to produce the final output [24, 23].

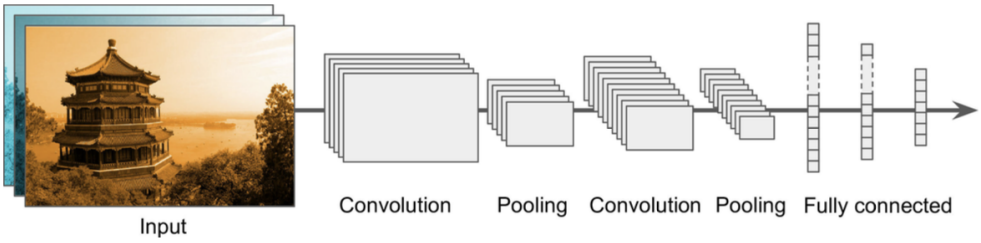


Figure 3.9: The schematic of a typical CNN from [26]

3.2.4 YOLO - You Only Look Once

The YOLO (You Only Look Once) architecture was developed by Joseph Redmon and Santosh Divvala in 2016. It's one of the most well-known single-shot object detection algorithms, not primarily for being the most accurate but for its remarkable speed at the prediction phase. The idea behind YOLO architecture is to utilize a localized feature analysis approach instead of examining the total image. YOLO works by splitting the input image into a grid, then predicting bounding boxes and class probabilities for each cell in the grid. The first goal of this strategy is to decrease computational requirements and enable real-time object detection: that's why it is particularly suited for applications where speed is critical, such as autonomous driving and real-time video analysis.
The YOLO algorithm follows these steps:

1. The input image is split into an $S \times S$ grid. Each grid cell is tasked with

detecting an object if the object's center lies within that cell.

2. For each grid cell, B bounding boxes and their associated confidence scores are predicted. A bounding box is described by five parameters: x, y, w, h, and confidence. x and y are the coordinates of the center of the bounding box in relation to the cell, w and h denote the width and height relative to the full image, and the confidence score indicates both the likelihood of an object being in that cell and the accuracy of the bounding box prediction.

3. Each cell estimates C conditional class likelihoods, representing the probability of an object belonging to specific classes, assuming an object is detected within the cell.

4. At test time, the probabilities of each bounding box containing objects from various categories are computed by multiplying the confidence score of each frame with the conditional class likelihoods. The final detection outcomes are obtained after applying threshold filtering and non-maximum suppression techniques.

The YOLO framework has evolved over time, with each new version enhancing its capabilities and addressing previous limitations. The most recent and sophisticated version, YOLOv8, was created by Ultralytics. This latest variant integrates state-of-the-art techniques to achieve exceptional results across multiple computer vision tasks. Beyond its core strength in object detection, YOLOv8 excels in precise instance segmentation and accurate image classification, showing its versatility and advanced feature set [28, 22, 29].

Insight on YOLOv8

The architecture of YOLOv8, or You Only Look Once version 8, is a state-of-the-art object detection model that advances the groundwork laid by earlier models [28].

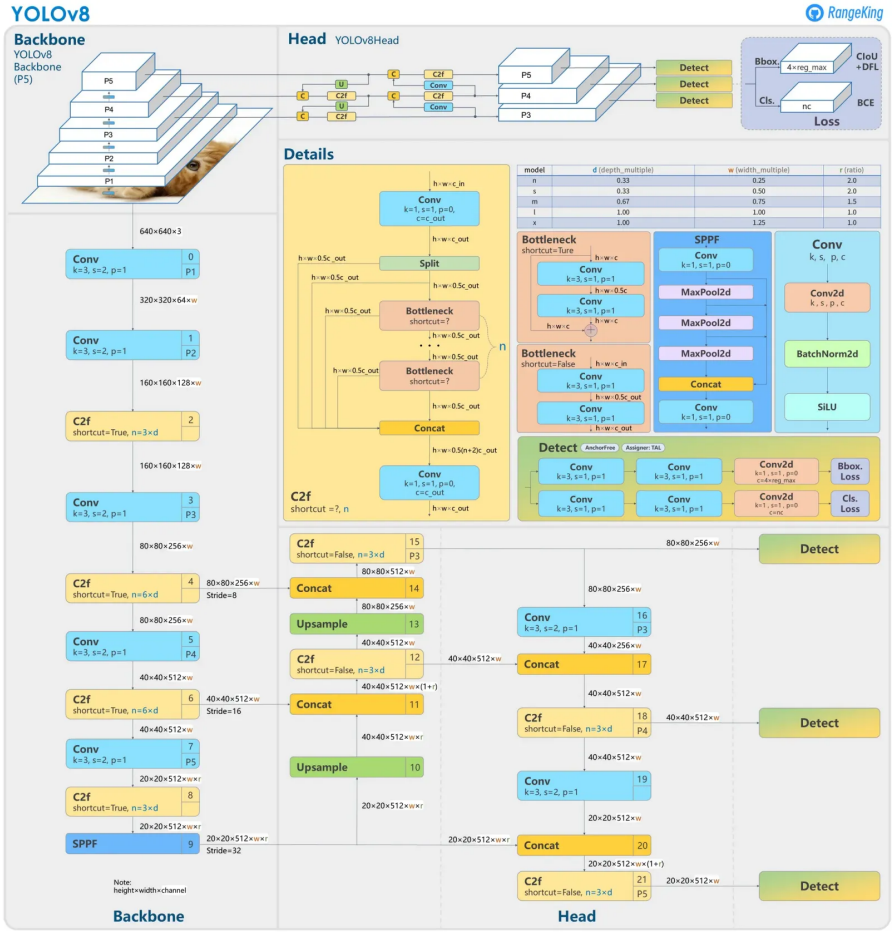


Figure 3.10: Architecture of YOLOv8 from [30]

As first big enhancements of YOLOv8 we can mention the modularity and the scalability of the architecture. The whole architecture can be split in three fundamental blocks: the backbone, the neck and the head. The first one aims at detecting the features from the input images. Then, the neck links the backbone with the head and allows the feature blending. Afterwards, the head's role lies in predicting the bounding boxes, object classes and confidence scores.
YOLOv8 offers the users the great opportunity to choose, among its variant, the one that best fits their exigencies, in terms of speed and accuracy. The variants vary from YOLOv8-tiny to YOLOv8x.
YOLOv8 introduces better training methodologies, including the implementation of Rectified Adam (RAdam) optimization and the option for anchor-based or anchor-free object detection. These updates lead to quicker training convergence and improved object detection capabilities.
Furthermore, YOLOv8 offers a versatile configuration system that allows users to

tweak lots of parameters, including input size, anchor boxes and model complexity, perfectly tailored to different datasets and application needs.
Now, let's go deep in the architecture of YOLOv8.

Backbone Network

The core of YOLOv8 architecture is the backbone network. It extracts hierarchical features from the input image, offering a detailed representation of the visual information. For instance, it's able to catch simple patterns in the first layers and features from different levels of abstraction [31]. The backbone used by YOLOv8 is CSPDarknet53, an evolution of the Darknet architecture: it allows the Cross Stage Partial networks, improving the learning skills and the efficiency.

Neck

YOLO uses a neck architecture, fundamental for feature fusion and concatenation. It links the backbone with the head, it provides multi scale information and deals well with the scalability of objects of different dimensions. What's more, it incorporates contextual information that helps in the detection task. Nevertheless, it decreases spatial resolution and usage of resources, boosting the speed of computation but compromising the model quality. The Neck gathers feature maps from various stages of the Backbone and it constructs a pyramid. To this aim, PANet (Path Aggregation Network), a feature pyramid network, is used by YOLOv8 [31].

YOLO head

The Head of the YOLO architecture is responsible for the network's output: the predictions. It forecasts bounding box coordinates, objectness scores, and class probabilities for each anchor box associated with a grid cell. The anchor boxes are used to detect objects of different shapes and sizes [31].

Chapter 4

The proposed Visual Servo architecture for Apple Harvesting

The goal of this thesis is the design of a visual servoing system for apple harvesting for a 6 DoF robot manipulator, equipped with an RGBD camera on its end effector. The proposed solution involves two main modules: one dedicated to the apples identification through a YOLOv8 Instance Segmentation model, while the second one focuses on the robot's motion control with Image-Based Visual Servo. In this chapter, a detailed description of the implemented methodologies will be provided. Firstly, an overview of the proposed architecture, along with its corresponding block diagram, is presented. Then, a delve into the fine-tuning and the dataset used to train the YOLOv8 CNN is done. Next, the implementation of the Image-Based Visual Servoing (IBVS) control strategy is outlined, followed by an in-depth of the pipeline implementation through a Finite State Machine.

4.1 Block Diagram and System Architecture

In the first place, some **assumptions** are needed to better frame the problem and explain the reasons behind implementation choices:

- the 6DOF robotic arm operates on a stationary basis
- the RGBD camera is mounted in an eye-in-hand configuration
- the pinhole model is the camera model employed to describe the mathematical relationship between the projection of 3D points onto the 2D image plane

- the targets are supposed to be static
- purely kinematic system: the dynamic of camera motion is neglected; it is assumed that the camera can precisely carry out the applied velocity control.

In light of these assumptions, we can now observe from 4.1 the block diagram of the proposed architecture, highlighting the workflow and the interaction among the subsystems: the robotic arm, the camera, the vision and control algorithms. The RGB-D camera, mounted on the manipulator's end-effector, represents a narrow sight from the robot's perspective. However, to enhance the robot to perceive the environment, the visual data are processed by the first main block of the architecture: the computer vision algorithm, implemented as a ROS2 node. By employing a YOLO v8 CNN, all apple instances are detected and segmented. Then, a policy is used to select the target to be reached. The extracted features of the target are passed to the Image-Based visual controller, which computes the control law to move the end effector toward the target. As soon as the tool reaches the closest identifiable (by the camera's depth sensor) distance, an open loop method is developed to successfully grasp the apple.

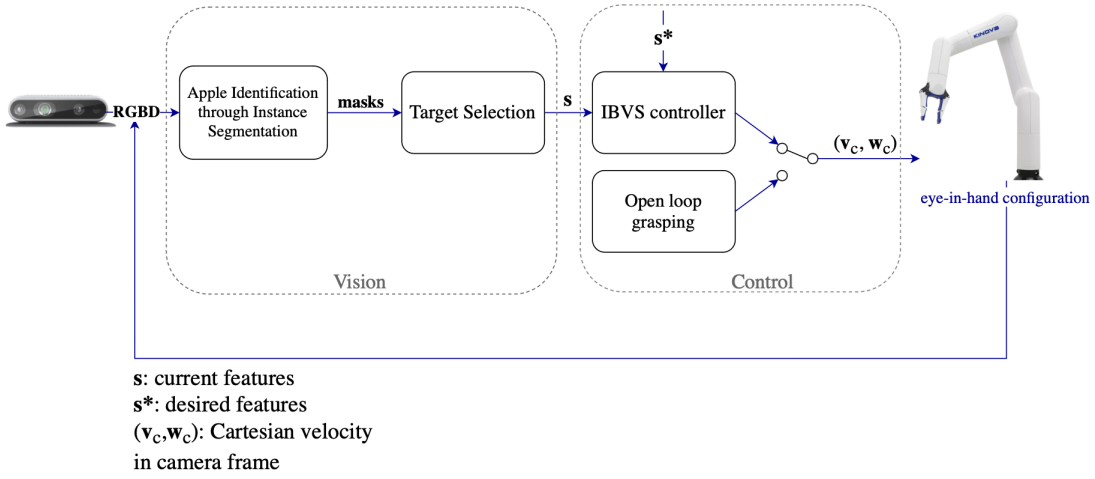


Figure 4.1: Block diagram of the proposed architecture

4.2 Vision block

The Vision Block in the proposed architecture is designed to identify the target, from the data stream provided by the RGB-D sensor, and extract relevant features to feed the visual controller. To accomplish this, a fine-tuned YOLOv8 model for apple instance segmentation has been integrated into the system. A detailed

explanation of the dataset and the fine-tuning process for training the network is presented, along with the cost function employed to select the optimal target from the detected instances.

4.2.1 The proposed methodologies: Instance Segmentation and Multi-class Classification

To address the challenge of visual perception for the manipulator, the selected computer vision task is the Instance Segmentation. Unlike basic object detection, instance segmentation offers more comprehensive and precise information about objects in the scene, including detailed insights into the location of them, allowing to orient the choice of the target apple to be grasped. For example, segmentation helps to distinguish apples based on their shape and degree of occlusion, providing insights into proximity and overlap with other objects, useful for the purpose of collision avoidance. To implement the Instance Segmentation model, we opted for YOLOv8 as efficient, accurate and fast state-of-the-art architecture, particularly well-suited to meet the real-time requirements of the IBVS controller.
The work involved training and fine-tuning two distinct models with the aim of selecting the one that demonstrated the best performance in validation and testing. The first network was tailored for detection and segmentation tasks only. The second model, in addition to detection and segmentation, was dedicated to classifying apples based on their level of occlusion, so as to obtain as output not only the mask and bounding box of the object, but also its class label, useful to guide the choice of the target while avoiding collisions.

4.2.2 Datasets

Let's now introduce the datasets used to train the YOLOv8 models.

- For the **Instance Segmentation** model, the dataset comprises a total of 1156 labeled images. Of these, 1064 images were provided by the PIC4SeR laboratory where the thesis was conducted, all captured in different apple orchards. This dataset is comprehensive, featuring images taken from various perspectives, under different lighting conditions, and with apples of different colors (red, yellow, combination of the two). Such diversity is crucial for ensuring proper training of the network. However, the dataset lacked close-up images of apples, which are essential for enabling the system to effectively perform the target approach operation. To address this gap, an additional 92 images were sourced online and labeled using the Roboflow platform. Furthermore, as the initial dataset included annotations in JSON format with apples categorized into different classes, Roboflow has been utilized to merge

all images, modify all classes into a single "apple" class, convert the dataset into YOLO format, and split it into: 809 images for training (70%), 232 images for validation (20%), 116 images for testing (10%). This dataset's split guarantees to assess the model performance on unbiased data.

- For the **Instance Segmentation and Multi-class Classification**, the aforementioned multi-label dataset, provided by the PIC4SeR, was used. This dataset comprised five classes: free ('free'), partially occluded ('partially'), heavily occluded ('heavily'), apple near an obstacle ('Obst_obst'), and apple near another apple ('Obst_apple'). This dataset was similarly converted into YOLO format using Roboflow and split into (70%) images for training, (20%) pictures for validation and (10%) images for testing.

In conclusion, the dataset is exhaustive in terms of lighting conditions, perspectives, and types of apples that can be found in an apple orchard. Nevertheless, the network is supposed to perform better in environments similar to apple orchards compared to other contexts.

Figure 4.2: Example of pictures from the train datasets

4.2.3 Fine-tuning and Training of the models

Below, the training and fine-tuning process is described.

- For the **Instance Segmentation** model training, the Grid Search technique has been used to fine tune the pre-trained YOLOv8s-Seg model. The "small"

model of YOLOv8 has been preferred as a trade-off, to guarantee better precision with respect to "nano" version, but to run faster than more precise versions like the "medium" one. The reason of exploiting Grid Search is due to the fact that the search space is limited and well-understood. Because of their crucial role in affecting the model's performance, the optimizer, the epochs and the learning rate have been selected, through the YOLO API, as parameters to be varied 4.1. Each hyperparameter influences different aspects of the training: the learning rate for dictating how quickly the model learns, the optimizer for controlling how the model's weights are updated and the epochs for establishing how many times the model is supposed to pass through the entire training dataset. Let's notice that the range of values for the learning rate was given to cover both conservative (small) and aggressive (larger) updates to the model weights; whereas the increasing number of epochs would have been effective to detect the occurrence of underfitting or overfitting.
Let's point out that, to increase the dataset variability and improve generalization, I've also applied some data augmentations (e.g., hsv_h, scale, fliplr), useful to simulate different lighting conditions, object scales and orientations.

- For the **Instance Segmentation and Multi-class Classification**, two types of trainings have been performed. However, instead of using Grid Search, a form of random selection among the learning rate, optimizer, and number of epochs was employed.

  1. Similarly to IS trainings, YOLOv8s-Seg was utilized as pre-trained model. However, since the learning process was conducted on the multi-label dataset, the model was learning about predicting the degree of occlusion the single detected and segmented apple was experiencing.
  2. The pre-trained models, to be fine tuned and trained, were selected from those exhibiting the best IS performances in validation. This second approach aimed to capitalize on the prior knowledge embedded in the pre-trained models. Indeed, this choice was motivated by the fact that the pre-trained weights for IS captured object boundary details and spatial relationships, then the model could more likely generalize to the occlusion classification problem.

Thus, basically, the main changes between the trainings of IS and IS with Multi-class Classification reside in the dataset the network is learning from and in the pre-trained model to fine tune.
As a final remark, the computational load was alleviated by the fact that the YOLO API allows you to explicitly set the training device to GPU. This significantly accelerated the training process, as the GPU is optimized for handling the parallel computations required for deep learning tasks.

Hyper-Parameter	Values Set
optimizer	{Adam, AdamW, SGD}
n. epochs	{100, 150, 200}
learning Rate	{0.0001, 0.001, 0.01}

Table 4.1: Hyper-Parameter Configurations explored via Grid Search trainings

4.2.4 Target Apple Selection

On the basis of the YOLOv8 model used for identifying the target apple, two different target selection policies have been developed.

- For the **Apple Instance Segmentation** model:

$$\text{target} = \arg\max_{i \in 1,\dots,N} A_i \tag{4.1}$$

  where N is the total number of detected apples, A_i is the area of the i-th apple' segmentation mask, target is the index of the selected apple.
  This method is straightforward and relies solely on the size of the detected apples in the image. The largest apple is selected as the target, which is based on the assumption that the largest visible apple is likely the closest or most prominent one in the scene. The policy's key points are the simplicity of implementation and the computational efficiency. As limitations, we can state that it doesn't consider occlusions or the quality (ripe enough or not) of the apple. By the way, it is well suited for our scenarios where all visible apples are of similar quality and the closest one is of interest.

- For the **Apple Instance Segmentation and Multi-class Classification** model:

$$\text{target} = \arg\min_{i \in \{1,\dots,N\}} C_i \tag{4.2}$$

  with

$$C_i = w_a \cdot \frac{1}{A_i} + w_c \cdot \text{cost}_{\text{class}}(L_i) \tag{4.3}$$

  Then:

  - N is the total number of detected apples.
  - C_i is the total cost for the i-th apple.
  - A_i is the area of the i-th apple's segmentation mask.
  - L_i is the class label of the i-th apple.

- w_a is the weight for the area-based cost (tunable).
- w_c is the weight for the class-based cost (tunable).
- $\text{cost}_{\text{class}}(L)$ is a function that returns the cost associated with class L, defined as:

$$\text{cost}_{\text{class}}(L) = \begin{cases} 0 & \text{if } L = \text{'Free'} \\ 1 & \text{if } L = \text{'Partially'} \\ 2 & \text{if } L = \text{'Obst\_apple'} \\ 3 & \text{if } L = \text{'Heavily'} \\ 50 & \text{if } L = \text{'Obst\_obst'} \end{cases}$$

This second selection policy is more sophisticated, combining information about both the size and the classification of each apple. It aims to balance the preference for larger apples with the desire to select apples that are less obstructed or more easily accessible. Indeed, the inverse of the area ($1/A_i$) is used so that larger apples have a lower cost contribution from this term. Different costs are assigned to various apple states (free, partially obstructed, etc.), allowing for intelligent selection based on accessibility: the high cost for 'Obst_obst' could effectively help in collision avoidance with other obstacles (leaves, branches, for examples). An other key point is the flexibility given by the weights w_a and w_c: they allow to adjust the relative importance of size versus classification.
To sum it up, the cost function approach offers several advantages: it can prioritize a slightly smaller but unobstructed apple over a larger but heavily obstructed one, it provides a way to avoid selecting heavily obstructed apples that might be difficult to interact with and it allows for fine-tuning of the selection process by adjusting the weights and class costs by experimentation, depending on the use case. However, it requires a more advanced model capable of both segmentation and classification. The effectiveness strongly depends on the accuracy of the classification model.

4.3 Visual servo block

The purpose of the Visual Servo block is to guide the robot's end effector to the desired position.
To fulfill the task, an Image-Based Visual Servo (IBVS) has been implemented, operating at $f_{controller} = 20Hz$. For feedback control, it exploits the visual features of the target, extracted by the Vision Module at an average inference speed of 5 fps ($f_{signal} = 5Hz$). The controller frequency $f_{controller}$ has been set to more than twice the inference frequency f_{signal} to ensure that the control action is continuously

updated with the latest data, thereby maintaining both stability and responsiveness. The system continues to adjust the robot's movement so that the target features (current) overlap with the desired features (set as reference) until convergence. Once the apple is sufficiently close that RGB-D sensor can no longer provide depth information, an open-loop strategy is employed to grasp the target. In the subsequent sections, the robot's motion control is accurately described.

4.3.1 Control Law Implementation

The IBVS controller was selected as the architecture for guiding the end-effector due to its robustness in handling the target's unknown and variable shape, real-time adaptability, flexibility in trajectory planning and ease of implementation. A key advantage of IBVS lies in its direct computation of errors based on feature values, without the need to reconstruct the apple pose.

Visual Features and Parameters

Let's clearly explain the visual features and feature parameters selected in this implementation, as well as the current and desired feature sets.

- **Visual Features:** Extracted by the vision data and computed by the YOLOv8 model, the visual features are the bounding box vertices of the segmented target apple. Beyond this set of points, another point has been added as visual feature: the center of the bounding box. Although three points are already sufficient to control the Kinova 6 DOF [13], a total of five points has been used to guarantee a more precise control.

- **Visual Feature Parameters:** The numerical values associated with the five points is trivially given by their coordinates (u, v), respectively the horizontal coordinate (in pixels, with the origin at the top-left corner) and the vertical coordinate (in pixels) of the points.

- **Current Features:** The current features represent the position of the target in the image plane, as captured by the sensor. Since the camera is mounted on the end-effector, the current features can update due to movements of either the target or the Kinova itself. However, in our project, the **target** is assumed to be **static** (e.g., stationary apples), so the current features update in the image as the robot moves.
  The current visual feature parameter vector is defined as:

$$\mathbf{s} = [u_1, v_1, u_2, v_2, u_3, v_3, u_4, v_4, u_5, v_5] \tag{4.4}$$

  where the index $i = 1 \dots 5$ refers to each point, with $i = 5$ representing the center of the bounding box, while $i = 1 \dots 4$ corresponds to the $i-th$ bounding

box vertex, ordered clockwise starting from the top-left corner (as shown in Fig. 4.3). The first four features are provided by the vision module, whereas the center of the detected object is computed by taking the average of the horizontal (u_1, u_3) and vertical (v_1, v_3) extreme coordinates of the bounding box (and rounding down the result to the nearest integer):

$$u_5 = \left\lfloor \frac{u_1 + u_3}{2} \right\rfloor, v_5 = \left\lfloor \frac{v_1 + v_3}{2} \right\rfloor \tag{4.5}$$

- **Desired Features:** The desired features represent the reference for the control system. In this scheme, they are kept constant during the operation because they relate to the position of the target that we aim to achieve with respect to the end-effector. Specifically, we want the target to be as close as possible to the tool while ensuring that the depth is captured by the sensor. The coordinate values were assigned as follows (Fig. 4.3): a mock-up apple was placed in front of the camera; the bounding box vertex values were printed to ensure the minimum distance from the target while capturing depth with the RGB-D camera (150 mm, experimentally determined) and the center coordinates were computed with 4.5. Thus, we have:

$$\mathbf{s}^* = [152, 70, 476, 70, 476, 393, 152, 393] \tag{4.6}$$

Naturally, for targets with shapes and sizes slightly different from the reference apple used in the experiment, the error may not converge exactly to zero. However, these variations are taken into account when assessing the error for convergence.

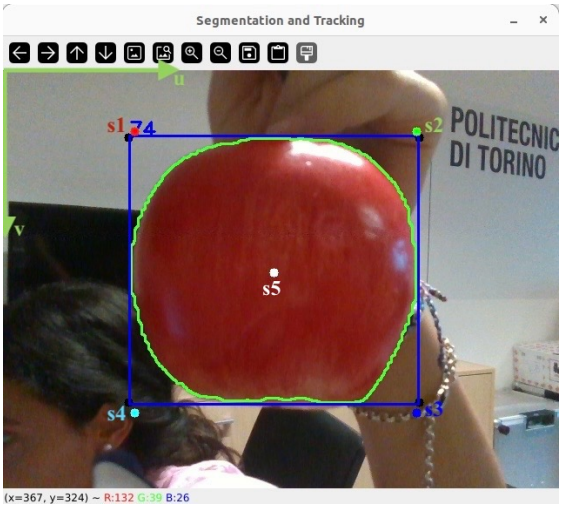


Figure 4.3: Desired visual features

Control law implementation

Conversion of the visual parameters from pixel to normalized image plane coordinates

Although the visual features coordinates are passed through the YOLO model in pixels, they are then converted into image plane coordinates. By assuming the pinhole camera model, for each feature, the conversion is done by applying the perspective projection equations 3.9:

$$x_i = \frac{(u_i - c_x)}{f_x}, y_i = \frac{(v_i - c_y)}{f_y} \tag{4.7}$$

where (x_i, y_i), with $i = 1 \dots 5$, are the normalized projected coordinates in the image plane, (u_i, v_i) are the pixel coordinates in the image, (c_x, c_y) represents the principal point of the camera (the image center), f_x and f_y are the focal lengths of the camera in pixels.

Error computation

After having converted the visual parameters of the detected target from pixels to image plane coordinates, the error is essentially calculated as the difference between the current feature and the desired feature:

$$\mathbf{e}(t) = [x_i(t) - x_i^*, y_i(t) - y_i^*]^\top \quad \text{for } i = 1, \dots, 5 \tag{4.8}$$

resulting in a 10-dimensional vector. Let's remark that t represents the time instant when the bounding box is updated by the vision module, $(x_i(t), y_i(t))$ are the current coordinates of the i-th feature point at time t, (x_i^*, y_i^*) are the goal's normalized image plane coordinates of Eq. 4.6, for the i-th feature point.

Update of the IBVS Control Action

If the target object is detected, the error is used for the control action computation, aiming at driving the tool toward the goal position.
As detailed in chapter 3, the IBVS control law outputs the Cartesian velocity, expressed in the camera frame, by means of the Interaction Matrix or feature Jacobian related to 4.4. This Interaction Matrix establishes the kinematic differential relationship existing between the camera velocity and the temporal variation of the error 4.8, that is how the visual features move as a result of the camera motion 3.3. Thus, the Interaction matrix $\mathbf{L_i}$ with $i = 1 \dots 5$ is constructed for each of the feature in this way:

$$\mathbf{L_i(t)} = \begin{bmatrix} -\frac{1}{Z_{centroid}} & 0 & \frac{x_i(t)}{Z_{centroid}} & x_i(t)y_i(t) & -(1 + x_i(t)^2) & y_i(t) \\ 0 & -\frac{1}{Z_{centroid}} & \frac{y_i(t)}{Z_{centroid}} & 1 + y_i(t)^2 & -x_i(t)y_i(t) & -x_i(t) \end{bmatrix} \tag{4.9}$$

The Jacobian matrix is time-variant: it updates as the visual features move in the image plane, then for any robot's motion captured by the Vision Module. If we compare 4.9 with the standard formula 3.18, we get that the factor f (focal distance) is canceled out due to the conversion of the pixel coordinates into the image plane coordinates.
Notice that, since the depth cannot be retrieved at the vertices of the bounding box of a rounded object (which would otherwise tend towards infinity), a reasonable choice is to approximate the depth of those points to that of the centroid of the segmented object ($Z_{centroid}$).
In the implementation, the centroid is considered as the arithmetic mean position of all the points in the mask shape provided by the CNN. Its coordinates are simply computed by averaging all x-coordinates and y-coordinates separately. So, given a contour represented by a set of points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ (output of the Vision Module), the centroid (x_c, y_c) is computed as:

$$\begin{aligned} x_c &= \frac{1}{n}\sum_{i=1}^{n} x_i \\ y_c &= \frac{1}{n}\sum_{i=1}^{n} y_i \end{aligned} \tag{4.10}$$

where n is the number of points in the contour and, x_i and y_i are the coordinates of the i-th point. In this way, we are assuming equal weighting for all contour points, which is appropriate for discrete point sets representing object boundaries.
Next, the overall interaction matrix $\mathbf{L}(t)$ is obtained by vertically stacking all the five visual features matrices:

$$\mathbf{L}(t) = \begin{bmatrix} \mathbf{L_1}(t) \\ \mathbf{L_2}(t) \\ \mathbf{L_3}(t) \\ \mathbf{L_4}(t) \\ \mathbf{L_5}(t) \end{bmatrix} \tag{4.11}$$

Now, to take into account the high variance in the depth measurement $Z_{centroid}$ recorded by the RGB-D sensor, we compute the Moore-Penrose pseudoinverse of the mean of $\mathbf{L}(t)$ and $\mathbf{L}_{des}$ matrices:

$$\mathbf{L_m}^{+}(t) = \frac{1}{2}(\mathbf{L}(t) + \mathbf{L}_{des})^{+}$$

where $\mathbf{L}_{des}$ is the overall Jacobian Matrix computed for the desired features and depth.

Finally, the IBVS control law aiming at nullifying the visual error is:

$$\mathbf{v}_{cf}(t) = -\lambda \mathbf{L}_{\mathbf{m}}^{+}(t)\mathbf{e}(t) \tag{4.12}$$

where $\mathbf{v}_{cf}(t)$ is the Cartesian camera velocity, including both the linear and rotational terms, and λ is the proportional gain scalar playing a role in the speed of convergence of the visual features to the desired ones. It was tuned by a trial-and-error procedure to balance the convergence (larger λ) and smoothness (smaller λ) of the action.

Notice that since the Kinova robot is supposed to receive Cartesian velocity commands in the tool reference frame, a conversion is needed:

$$\mathbf{v}_{tf} = \mathbf{R}_{tc}\mathbf{v}_{cf} - \mathbf{R}_{tc}(\omega_{cf} \times \mathbf{t}_{tc})$$
$$\omega_{tf} = \mathbf{R}_{tc}\omega_{cf}$$

where $\mathbf{v}_{tf}$ is the linear velocity in the tool frame, ω_{tf} is the angular velocity in the tool frame, $\mathbf{v}_{cf}$ is the linear velocity in the camera frame, ω_{cf} is the angular velocity in the camera frame, $\mathbf{R}_{tc}$ is the rotation matrix from the camera frame to the tool frame, $\mathbf{t}_{tc}$ is the translation vector from the camera frame to the tool frame. The dependence on t was omitted for seeking simplicity. In the implementation, the matrix $\mathbf{R}_{tc}$ and the vector $\mathbf{t}_{tc}$ are extracted by the TF2 module of ROS2.

Smoothing methods

Either any inconsistency of the inference on the detected visual features or the big variance on depth value can impact a lot on the control action $\mathbf{v}_{cf}$, causing sudden changes or noise in the velocity command. Thus, as first mean to smooth and stabilize the camera movement, the EMA (Exponential Moving Average) filter is used:

$$\mathbf{v}_{cf} = \alpha\mathbf{v}_{cf}(t) + (1-\alpha)\mathbf{v}_{cf}(t-1)$$

where $0 \leq \alpha \leq 1$ is the smoothing factor. Instead of directly inferring the new velocity, this is combined with the velocity computed one iteration before. The α factor balances the weight of the currently computed velocity over the previously filtered one. When α is closer to 1, the filter responds more quickly to new changes, while, when α is closer to 0 the filter responds more slowly to new changes and more smoothing occurs, reducing noise but potentially introducing lag. The α has been tuned by trial-and-error procedure trying to find a balance between responsiveness and smoothness.
Moreover, to further reduce the motion oscillation, a saturation value for the velocity has been set, tuned during experiments.

4.3.2 Transition from Closed-Loop to Open-Loop Control

Let's suppose that the current features are correctly converging to the desired ones. An issue arises when the target object is closer to the camera than the

minimum detectable range of the RGB-D sensor: the depth $Z_{centroid}$ is recorded as 0. Reasonably, the sensor is no more able to measure the distance to the object. This implies a singularity in the interaction matrices $\mathbf{L}(t)$, leading to an infinite values for the camera velocity, thus an unstable control behaviour.

By assuming the proximity to the target when the acquired depth $Z_{centroid}$ is lost, the switching to an other control strategy is necessary to grasp the apple. The proposed solution aims at overcoming the limitation of the visual controller by implementing an open loop method. The reasons leading to this choice are: the effectiveness of this mechanism since it is triggered in the precise positioning achieved by the visual servoing phase that precedes it and the simplicity in its implementation. An alternative to this would have consisted in reconstructing the pose and passing it to the MoveIt2 planner to plan the grasping. Nevertheless, this method increases the computational complexity.

In the implementation of the open loop, the constant Cartesian linear velocity commands were given to the robot with the intention of accounting for the misalignment between the origin of the tool reference frame and the origin of the camera reference frame as shown in Fig. 4.4:

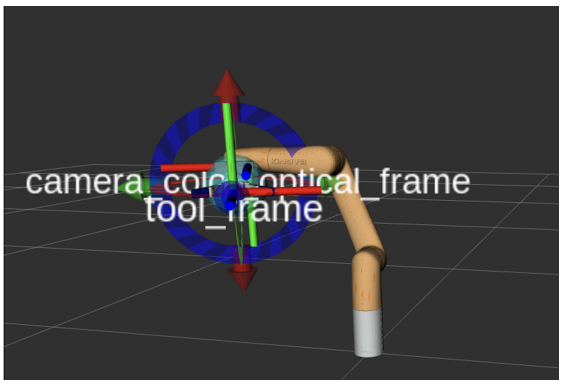


Figure 4.4: Representation of the misalignment between the camera frame (camera_color_optical_frame) and the tool frame

- the commands are sent to the robot for a fixed open loop duration;
- the centroid of the detected bounding box is converted from pixels to camera frame, by using 4.7 and multiplying the resulting coordinates in the image frame by the distance to the center of the apple, experimentally measured $Z_{centroid}$=159 mm. The z coordinate value of the centroid in the camera frame is trivially $Z_{centroid}$=159 mm. Then, by calling TF2, the centroid is converted to the tool frame, obtaining the 3D coordinates $(\mathbf{x}_{tf}, \mathbf{y}_{tf}, \mathbf{z}_{tf})$;

- by applying the equations for uniform rectilinear motion, we send the following Cartesian commands to the robot in the tool frame:

$$\mathbf{v}_x = \frac{\mathbf{x}_{tf}}{\Delta_{OL}}, \mathbf{v}_y = \frac{\mathbf{y}_{tf}}{\Delta_{OL}}, \mathbf{v}_z = \frac{\mathbf{z}_{tf}}{\Delta_{OL}}$$

In the implementation, the fixed open loop duration Δ_{OL} is set for how long the constant velocity is applied. At the end, the gripper is closed by calling a ROS2 server.

4.4 Finite State Machine of the system

A comprehensive and integrated pipeline has been developed to evaluate the individual performances of the two modules and their combined efficacy.
The overall system has been modelled as a Finite State Machine (FSM), shown in Fig. 4.5, dictating the robot's actions. It has been implemented in ROS2 with Python.

The system enters the FSM in the `WAIT_FOR_TARGET` state, where it waits for a target to be detected. Meanwhile, a specific ROS2 node does the subscription to the topic `/camera/color/image_raw` containing the RGB images recorded by the camera sensor. Then, the fine-tuned YOLOv8 CNN inferences the visual data, outputting the mask of pixels for each detected apple instance. According to the selection policy, a single apple is chosen as target and its extracted features (the bounding box vertices) and the computed centroid are published respectively on `/bounding_box_target` and `/centroid_target`. If no target is found, the system stays in `WAIT_FOR_TARGET` state. Otherwise, if the target apple is found (e.g. if any element of the bounding box array is non-zero), the system is triggered to pass to the `START_CONTROLLER` state, where it checks the depth value of the centroid `/centroid_target` from the topic `/camera/aligned_depth_to_color/image_raw`.
If $Z_{centroid}$ of the detected target is greater than 0, the target is supposed to be far so the system enters the `CLOSED_LOOP` state where the IBVS control law is computed and the Cartesian velocity commands are published on the `/twist_controller/commands` topic.
Either from the `START_CONTROLLER` state or `CLOSED_LOOP` state, if $Z_{centroid}$ is measured as 0, the object is considered to be near and the system enters the grasping sequence, switching to the `OPEN_LOOP` state, where the linear velocity commands are published on `/twist_controller/commands` topic for a given open loop duration. Afterwards, the gripper is closed by calling a ROS2 server, and the grasping sequence continues with the `PLACE` state, to drop the object by opening the gripper, and ends with the `RETURN_HOME`. Both the motion plans to the placement pose

and home pose are executed by ROS2 servers utilizing the `MoveGroupInterface`, the MoveIt 2 class enabling to control the `manipulator` move group and plan a motion.
Let's remark that during the grasping sequence (`OPEN_LOOP`, `PLACE`, `RETURN_HOME`), the system executes the state subsequently. Some checks are performed to avoid the system to leave the sequence: during the open loop, for example, the visual features of the target are expected to be lost. For this purpose, the method `execute_state_machine()`, aiming at handling the states transition, switches to states `WAIT_FOR_TARGET`, `START_CONTROLLER` or `CLOSED_LOOP` only if the system has not already entered the grasping sequence. Finally, the `RETURN_HOME` state is followed by the `WAIT_FOR_TARGET` state again, ensuring a continuous harvesting process.

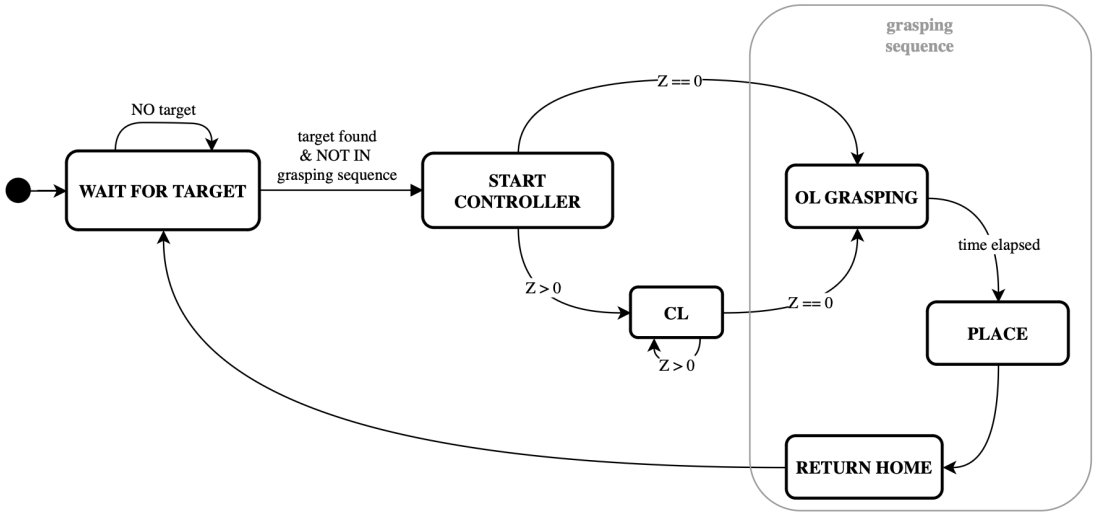


Figure 4.5: Representation of the Finite State Machine of the system

Chapter 5

SW and HW tools

This chapter introduces the key components to simulate, test and implement the robotic system developed in this thesis work. It begins with a brief overview of robotic systems, setting the stage for a detailed discussion of the specific hardware and software tools. The hardware section covers the Kinova robotic arm and the Intel RealSense visual sensor. The software section discusses about ROS2, Gazebo and MoveIt2. The chapter concludes with an overview of machine learning tools, including YOLOv8 API and OpenVINO.

5.1 Introduction to robotic systems

A robotic system is an integrated assembly of mechanical, electronic and computational components designed to perform tasks autonomously or semi-autonomously. These systems are composed of both hardware (like actuators, sensors, motors, links and joints) and software elements (comprising the operating system, control algorithms, simulation tools, ML models) working together to enable the robot to sense, process, and act in its environment. Robots vary their architecture and function according to the needs of users and applications, with common configurations including:

- *Industrial Robots*: Used in manufacturing for tasks like assembly, welding, and painting. These robots are typically stationary and operate in highly structured environments.
- *Unmanned Aerial Vehicles*: Drones are an example of UAVs. These robots are used for aerial surveillance, mapping, and even delivery services.
- *Humanoid Robots*: Robots designed to mimic human movement and interaction, used in research, customer service and assistive technologies.

- *Mobile Robots*: Robots capable of moving through their environment, such as autonomous vehicles or delivery robots.

In this project, the Kinova Gen3 Lite 6-DOF, equipped with an Intel RealSense d435i camera as RGBD sensor on its end effector, represents the manipulator where the visual servoing has been designed on.

5.2 Kinova Gen3 Lite

The Kinova Gen3 Lite is a versatile and lightweight robotic arm designed for collaborative operations requiring precision, flexibility and ease of integration. As a six degrees of freedom (DOF) manipulator composed solely of revolute joints, it is well-suited for tasks in research, education and light industrial automation.

Figure 5.1: Kinova Gen3Lite 6 DOF from [32]

Fig. 5.2 depicts the schematic of the Kinova Gen3 Lite manipulator with all the dimensions and reference frames shown. Let's remark that all revolute joints are at 0 position. We can notice that the Kinova Gen3 Lite Gripper, with two fingers, is attached to the end of the manipulator, allowing it to move and manipulate objects.

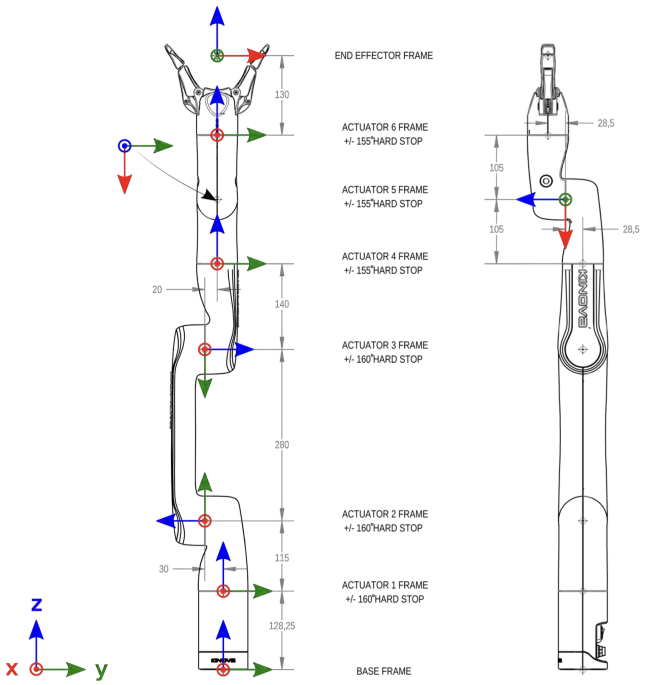


Figure 5.2: Kinova Gen3 Lite schematic with frame definitions and dimensions in mm from [33]

The arm features a wide range of sensors like position, current, voltage, temperature, accelerometer, and gyroscope, enhancing its operational awareness, adaptability and safe interaction with its environment. It is equipped with actuators and an embedded controller, ensuring smooth operation with both high-level and low-level control options. The manipulator operates with a 1 kHz low-level, closed-loop control system, providing real-time feedback and adjustment. The Gen3 Lite runs on the Kinova® Kortex™ API software. The connectivity is offered through USB, Ethernet, and RNDIS (Remote Network Driver Interface Specification). Furthermore, users can access the Kinova Web App from any desktop, laptop, or mobile device, making programming and control more efficient and accessible. The Gen3 Lite integrates with ROS (Robot Operating System), MATLAB, C++ and Python, offering robust software support for tasks such as motion planning and object manipulation.
Additionally, its modular architecture allows for easy customization, enabling users to adapt the robot to specific needs. In Tab. 5.1, the general technical specifications have been listed and the sensors connected with the robot have been reported [32].

Specification	Value
Weight	5.4 kg
Payload	500 g
Maximum Reach	760 mm
Degrees of Freedom	6
Max Cartesian Translation Speed	25 cm/s
Actuator Joint Range	+/- 155 to 160°
Power Supply Voltage	18 to 30 VDC, 24 VDC nominal
Average Power	20 W
Ingress Protection	IP22
Operating Temperature	0 °C to 40 °C
Sensors	Position, current, voltage, temperature, accelerometer and gyroscope

Table 5.1: General Specifications of Kinova Gen3 Lite

In table 5.2, the interfaces of Kinova Gen3 Lite have been presented.

Interface	Details
Software	KINOVA®KORTEX™
Internal Communications	1 x 100 Mbps Ethernet
API Compatibility	Windows 10, Linux Ubuntu 18.04, ROS Melodic
Programming Languages	C++, Python
Basic Interfaces	USB-A, micro USB, Ethernet, Wi-Fi
Control System Frequency	1 kHz
Low-Level Control	Position, velocity, current
High-Level Control	Cartesian position/velocity, joint position/velocity

Table 5.2: Interfaces of Kinova Gen3 Lite

Since the velocity commands to the Kinova robot will be sent in Cartesian mode in this thesis work, table 5.3 is particularly useful to get acquainted about the Cartesian Limitations.

Limit		Value
twist limits	linear	0.25 m/s
	angular	45.8°/s (0.80 rad/s)

Table 5.3: Kinova Gen3 lite Cartesian limitations

5.2.1 Introduction to Kinova Gen3 Lite Operation

It's possible to connect the Kinova Gen3 Lite robot to a computer via [33]:

- USB (RNDIS) connection, where the robot's DHCP server automatically assigns an IP address to the computer, enabling Ethernet;
- USB Type-A to Ethernet Adapter, allowing the robot to be connected over a local area network (LAN);
- Wi-Fi, where any device on the same network can communicate with the robot wirelessly using the assigned IP address.

The Kinova Gen3 Lite robot can be controlled in three different ways through the PC [33]:

1. using a physical gamepad, like an Xbox controller;
2. through virtual joysticks over a network connection via the KINOVA® KORTEX™ Web App;
3. programmatically via the KINOVA® KORTEX™ API (that is how it occurs in this project).

KINOVA® KORTEX™ Web App

The KINOVA® KORTEX™ Web App is a user-friendly, web-based graphical interface that allows users to interact with and control the Kinova Gen3 Lite robot without needing to write any code.
Among the key features of the KORTEX™ Web App we can find: the real-time control of the robot through various modes, including virtual joysticks; the possibility to configure robot's performance settings, safety thresholds, and protection zones; the monitoring of robot's system parameters, including sensor data, joint positions, and operational status; the support to firmware updates [33].

KINOVA® KORTEX™ API

The KINOVA® KORTEX™ API is a powerful tool-set that enables developers to programmatically interact with and control the Kinova Gen3 Lite robot. In a structured way, this API provides the access to robot's functionalities, advanced customization, automation, and integration of the robot into larger robotic systems or applications.
Among the essential characteristics of the KINOVA® KORTEX™ API we can mention: the control of the robot both in Cartesian and Joint-Level commands; the support for Multiple Programming Languages, including C++ and Python;

Asynchronous and Synchronous Operations; Low-Level and High-Level Control [33].

KINOVA® KORTEX™ GitHub Repository

The KINOVA® KORTEX™ GitHub Repository is an open-source resource provided by Kinova, where developers can access code examples, libraries, and tools related to the KORTEX™ API. The repository includes: a variety of code examples covering common tasks such as moving the robot, reading sensor data and implementing safety features; pre-built libraries and Software Development Kits (SDKs); extensive documentation that explains how to use the API; packages and examples for integrating the KORTEX™ API with ROS.
KINOVA® KORTEXTM ROS is the repository that contains ROS packages to interact with Kortex, simulate and control the robot. Additionally, it provides support for Gazebo and MoveIt as well. To read more about this, we suggest the reader to consult [34].

5.3 Intel RealSense D435i camera

Visual sensors are critical components in robotics and computer vision, enabling systems to perceive and interpret their surroundings: they are able to capture not only RGB images but also depth information. The Intel RealSense D435i has been used in this thesis to provide 3D perception.

Figure 5.3: Intel RealSense D435i from [35]

The D435i combines high-resolution depth sensing with an integrated inertial measurement unit (IMU), offering both visual and motion tracking capabilities [35]. The depth part utilizes active stereo vision technology, where two infrared cameras work together with an infrared projector to capture depth information. This setup allows the camera to generate detailed depth maps. The global shutter feature of the camera ensures the accurate capture of depth data, even when objects are in motion or the scene is illuminated by different lighting conditions. The inclusion of an integrated IMU enhances depth sensing capabilities of the D435i by providing additional data on motion and orientation, which is critical for applications like SLAM (Simultaneous Localization and Mapping). Developers can leverage the

RealSense SDK, which provides a powerful API to access and manage depth data, RGB images, and IMU readings. The D435i is also fully compatible with ROS2, making it an ideal choice for integrating 3D perception into robotic systems [35].

General information	
External dimensions ($L \times W \times H$)	90 × 25 × 25 mm
Ideal Range	0.3 to 3 m
Use environment	Indoor/Outdoor
Depth	
Depth technology	Stereoscopic
FOV	87° × 58°
Resolution	Up to 1280 × 720
Frame rate	Up to 90 fps
Depth Accuracy	< 2% at 2 m
Minimum Depth Distance at Max Resolution	28 cm
RGB	
RGB technology	Rolling shutter
FOV	69° × 42°
Resolution	Up to 1920 × 1080
Frame rate	Up to 30 fps
Sensor resolution	2MP

Table 5.4: Technical specifications of Intel Realsense Depth Camera D435i

5.4 Robot Operating System

ROS2, the second generation of the Robot Operating System (ROS), represents a significant advancement in the field of robotics, addressing the growing need for more robust, flexible and scalable robotic systems. It was created to overcome the limitations of its predecessor, ROS1. As Fig. 5.4 shows, ROS2 represents a robot programming middle-ware (meta-operating system), serving as a layer of software that operates between the operating system (OS) and user applications [36]. This middle-ware facilitates the development and execution of robots by providing essential tools and libraries, as well as a development methodology that supports the creation, integration and monitoring of robotic applications. Specifically, ROS2 includes communication mechanisms for distributed components, compilation systems and monitoring tools, all of which are open source. Furthermore, ROS2 incorporates a rapid data distribution service, adheres to real-time requirements and is compatible with various operating systems, like Linux and Windows. The primary programming languages used are C++ and Python.

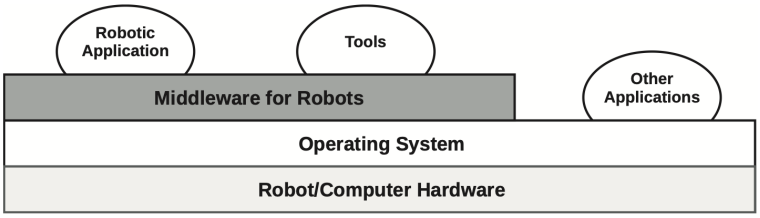


Figure 5.4: Depiction of software layers within a robotic system from [36]

The software packages in ROS2 are developed and maintained by a global community of developers, including organizations, researchers and hobbyists. These packages are published in online repositories, most notably on GitHub, where they are freely accessible to the public.
The packages in ROS2 are organized into distributions, which are collections of packages designed to work together, ensuring stability and compatibility within a given version [36]. In this thesis project, ROS2 Humble has been used as distribution.

Architecture of ROS2

ROS2 is a middle-ware that uses a strongly-typed, anonymous publish/subscribe model, enabling message exchange among various processes. Any ROS 2 application can be represented by a Computational Graph, showing the network of its main components, the Nodes, within the system and the communication links between them. Let's now delve into the definition of the fundamental core of ROS2, the node, and into the paradigms of communications among the nodes [36].

Nodes

The fundamental computational units in ROS2 are referred to as Nodes. They can be implemented either in C++ or Python. We essentially define the node as an instance of the Node class [36]. The execution of a node can be categorized into two primary modes:

- **Iterative execution**: Here, the node configures a timer linked to a callback function. The callback function is invoked at a predefined frequency, when it executes a specific control task.

- **Event-driven execution**: As suggested by the name, in this case, the node reacts to events as they occur, then it associates a callback with asynchronous events.

The paradigms of communication in ROS2

In ROS2, a node can communicate with other nodes according to three different paradigms [36].

1. **Publication/Subscription**: In an asynchronous way, nodes can publish messages to a topic that reaches its subscriber. If the messages are data with a well-known structure, specified in a file with .msg extension, the topic serves as a communication pathway that allows nodes to interact with one another in a uni-directional manner. Within a node, you can define publishers and subscribers for a topic. A publisher is a node that transmits messages to a designated topic, whereas a subscriber is a node that listens for messages from that specific topic. Multiple nodes can subscribe to the same topic and, whenever a message is published, all subscribers will execute their corresponding callback functions [37].

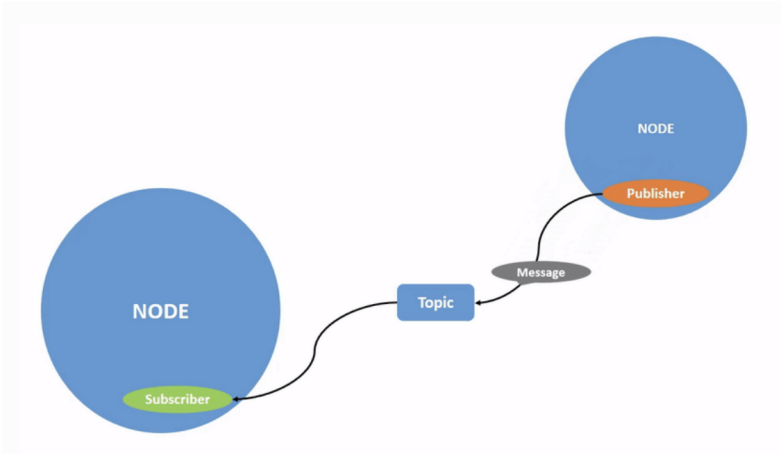


Figure 5.5: Representation of the publisher/subscriber communication paradigm through topic from [37]

2. **Services**: Services are another paradigm of communication among nodes: they involve synchronous mechanism where a node (called client) sends a request to another node (server) and waits for a response. This type of communication interface is specified in a file with the .srv extension. Typically, this bi-directional communication demands a prompt reply to avoid disrupting the control cycle of the node that initiated the service call [37].

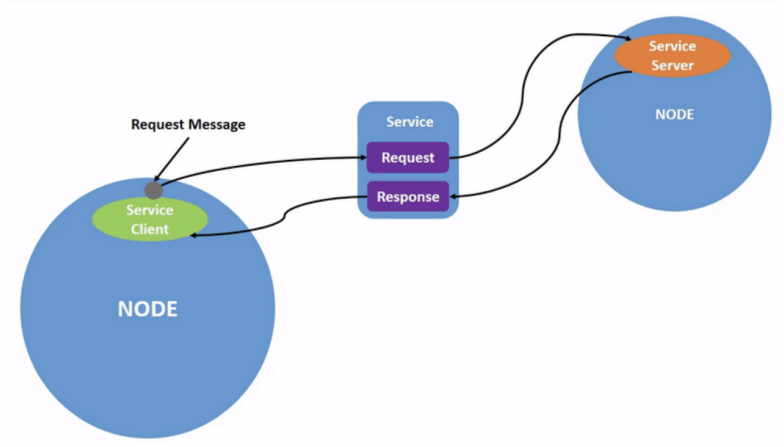


Figure 5.6: Representation of the client/server communication paradigm through service from [37]

3. **Actions**: This third paradigm is well-suited for long-running tasks. Based on a client-server mechanism, the actions present asynchronous communication. The client sends a remote procedure call (the goal) to the action server and it waits without any blocking state. While completing the task, the action server will periodically publish feedback on a topic about its status. Finally, the server will reply by sending a result. The file with extension .action contains all the three components: the Goal, the Feedback and the Result [37].

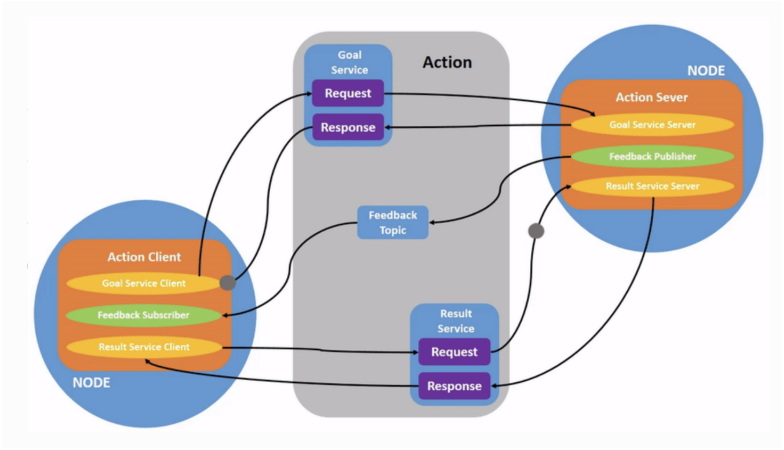


Figure 5.7: Representation of the client/server communication paradigm through action [37]

Transform library Tf2

Tf2 is an efficient and flexible tool in ROS2, designed to keep track of coordinate frames and manage their transformations over time.

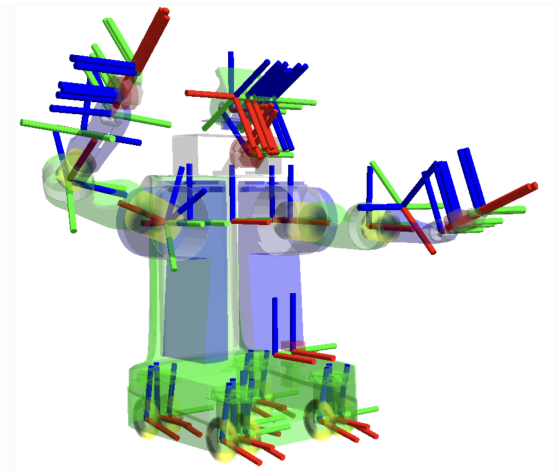


Figure 5.8: Representation of RFs in a robot with Tf2 from [38]

Tf2 operates by storing a series of transformations that describe the positions and orientations of different Reference Frames (RFs) relative to each other. These transformations are managed in a hierarchical structure called transformation tree, which is continuously updated as the robot and its environment change over time. Each RF is allowed to have only one parent but can be connected to multiple child frames. If a reference frame is linked to more than one parent, the system will generate an error.
When a specific transformation is required by a node, the node uses TFListeners. These objects keep a buffer that stores the most recent transformations and offer an API that can [38]:

- check if a transformation exists between two reference frames at a specific time;
- retrieve the rotation or translation between two reference frames at a given moment;
- convert a coordinate vector from one reference frame to another at a particular time.

One of the key features of Tf2 is its ability to handle transformations across both space and time. This means it can account for the fact that the robot and objects in its environment may be moving, and it can interpolate or extrapolate

transformations to provide accurate spatial information at any given moment (also prior to the time of request). Additionally, if two reference frames, e.g. A and B, are not directly linked, Tf2 can automatically manage the necessary matrix operations using any intermediate frames that connect them [38].

5.5 Gazebo

The availability of frameworks for the development of robotic systems is as fundamental as the availability of robotics simulation tools, allowing us to test the system in a virtual environment, prior to implementation on physical hardware. Gazebo is a powerful open-source simulation tool for creating and testing robotic systems. The framework models both indoor and outdoor scenarios, providing an artificial environment where robots can be tested under various conditions: complex terrains, lighting and sensor noise. Additionally, it excels in supporting advanced physical models, allowing for simulations of dynamics and sensor data. Users can use the Gazebo API, enabling programmers to extend and tailor the simulator by implementing new plug-ins (for custom sensors or robot models). Programmers can create their own worlds and robot models by using the SDF (Simulation Description Format) and URDF (Unified Robot Description Format) languages. Both the languages are XML-based format: the first one aims at describing world properties like physical characteristics, lighting, sensors; the second one is designed for describing the physical and visual aspects of robots, such as joints, links and sensors.
One of the key advantages of Gazebo is its integration with ROS/ROS2. Through this synergy, Gazebo and ROS allow to identify potential system issues, optimize performance and validate the design [39].

5.6 MoveIt 2

In addition to simulation tools, frameworks for robot motion planning and manipulation are essential for developing advanced robotic systems. MoveIt 2 is a state-of-the-art, open-source software framework that provides tools for robotic motion planning, manipulation, 3D perception, kinematics, control and navigation [40]. MoveIt 2 supports a wide range of capabilities, including path planning, collision detection, inverse kinematics, and trajectory generation.
Users can interact with MoveIt 2 through its comprehensive API, which enables the customization and extension of its functionality to meet specific project needs: custom motion planners, new sensors or modified kinematic model of a robot can be integrated.
MoveIt 2 supports both SRDF (Semantic Robot Description Format) and URDF

formats for describing the robot's physical and kinematic properties. In particular, SRDF complements the URDF, expliciting joint groups, robot configurations, collision checking information and any transforms useful to express the robot's pose.
Like Gazebo, also Moveit 2 provides the integration with ROS/ROS2 [40].
In this thesis project, MoveIt 2 Humble has been used as distribution.

moveit servo

It's a Moveit package that allows the user to send realtime end effector velocity commands to the manipulator.
To use this high-powered tool, it's only needed to have: URDF and SRDF files of the robot, a controller receiving joint positions or velocities as input and joint encorders.
The commands can come from anywhere: joystick, keyboard or other controllers.
The main features of moveit servo include:

- the ability to control the robot both in the task space (Cartesian End-Effector twist commands) and in Joint space;
- the Collision checking to prevent unsafe motions;
- the Singularity checking;
- the imposition of Joint position and velocity limits;
- the possibility to send inputs as simple ROS messages.

Moveit Servo is very flexible in its development, since it can be launched as a "node component" or a standalone node [41]. Servo can be included in a node through the C++ interface as well.
This package has been used to do virtual simulation of the IBVS on the Kinova robot.

5.7 YOLOv8 API

The YOLOv8 API offers a comprehensive and user-friendly approach to developing object detection, instance segmentation, classification models, ecc.
As first step, the data preparation is needed. The API expects data to be organized in a specific structure, usually defined by a YAML file. This file acts as a roadmap, pointing to the locations of training, validation, and optional test sets. Each set should contain images along with their corresponding label files, typically in a simple text format.
Once the dataset is ready, the API provides a suite of methods to interact with

the model. The training process is initiated through a dedicated method, which allows for extensive customization. First of all, YOLO offers the possibility to fine-tune a pre-trained model. All the YOLOv8 segmentation and detection models have been pre-trained on the dataset COCO, while all the YOLOv8 classification models have been pre-trained on the dataset ImageNet. The user can select a pre-trained model from the smaller but faster version (for example, in the specific case of Instance segmentation: YOLOv8n-Seg) to the bigger but more accurate one (for example, in the specific case of Instance segmentation: YOLOv8x-Seg). Users can then fine-tune various hyper-parameters to suit their specific needs, including: the number of training epochs, batch size, input image dimensions, learning rate, ecc. The API also supports data augmentation techniques, which can be adjusted to increase the model's ability to generalize.
Validation is another aspect handled by the API: it can be run as a standalone process or integrated into the training loop. Validation also allows for parameter adjustments.
After training and validation, the API facilitates easy testing and prediction on new data. Users can apply their trained models to individual images or entire directories, with control over confidence and IoU thresholds.
Throughout these processes, the API generates a set of results and metrics. During training, users can monitor various loss values, learning rate progression, and even visualize training images. Validation results provide critical insights into model performance, including mean average precision (maP) at different IoU thresholds, precision and recall values, and confusion matrices. The API supports the generation of precision-recall curves and feature maps.
Model management is another suit of the YOLOv8 API: it offers straightforward methods to save trained models and load them later, facilitating further fine-tuning [29].

5.8 OpenVINO: Accelerating AI Inference

OpenVINO (Open Visual Inference and Neural Network Optimization) is an open-source toolkit developed by Intel, designed to optimize and accelerate the inference of deep learning models across a variety of Intel hardware platforms (including CPUs, integrated GPUs, FPGAs, and VPUs). By converting a model from frameworks such as YOLO to the OpenVINO format, you can harness its powerful capabilities to improve performance and efficiency.
The toolkit includes a Model Optimizer, converting models from popular frameworks into an intermediate representation (IR) format optimized for inference, and an Inference Engine, executing these optimized models on the target hardware [42].

5.9 Roboflow

Roboflow is a great framework offering tools for each step of computer vision pipeline.
Roboflow excels in data management for computer vision projects. It provides equipments for uploading, organizing, and versioning datasets. Users can easily import images and annotations from various sources and formats. Then, lots of robust preprocessing options are available: image resizing, augmentation techniques and annotation format conversions. In this way, dataset can be suited for different model architectures.
Furthermore, Roboflow allows users to personally annotate data [43].
In this thesis project, both the data version exportation and the annotation tools have been exploited to adapt the dataset to YOLOv8 architecture input's requirements and to draw the mask of apples in some unlabeled pictures.

Chapter 6

Results

The following chapter presents a comprehensive analysis of the experimental results, focusing on the performance of the YOLOv8-based Convolutional Neural Network for instance segmentation, the efficacy of the Image-Based Visual servo and the overall system architecture.

6.1 Validation process of the Vision model

The objective of this phase was to select the optimal CNN model, from among those generated during the training process, to perform inference on the RGB data stream.
The validation process for both the Instance Segmentation (IS) and IS and Multi-class Classification (MC) models comprised three distinct stages:

1. **Initial Analysis**: All resulting models underwent a preliminary examination. The primary screening criteria included:

   a) Assessing for overfitting by discarding both models whose validation loss function didn't remain above the training loss function and models with increasing loss function. In this way, we assured that the models weren't memorizing the training data but were learning generalizable features.

   b) Evaluating performance using the mean Average Precision (mAP) metric, defined in Appendix D. In particular, for gauging YOLOv8 performances, the mAP50-95 has been employed: it computes mAP at varying IoU thresholds, ranging from 0.50 to 0.95, making it more robust compared to mAP50 because it assesses the model's accuracy and detection capabilities across a broader range of precision levels.

2. **Prediction on Unseen Data**: The networks' performance was further assessed by observing the prediction on unseen images and test videos, showing

the network's generalization capabilities and robustness when coping with novel data.

3. **Laboratory Testing**: Real-time inference tests were conducted in a laboratory setting using a mockup apple. These tests evaluated the models' performance under two conditions: with a stationary apple and with the apple moving within the camera's field of view.

These tests revealed the necessity for model optimization to speed up the real-time inference: the YOLOv8 models were converted to OpenVINO format, reaching an average inference rate of 5 fps.

6.1.1 Instance Segmentation model results

Tab. 6.1 shows the results from Grid Search training for the IS model in terms of precision, recall, mAP. We can observe that the metric values are not too far apart for each fine-tuned model. (B) refers to "best" performance metric achieved by the model during the training process, e.g. mAP50-95(B) stands for best mean average precision at an IoU threshold of 0.5-0.95 achieved by the model during training.
However, three models were selected based on the criteria of the **initial analysis**, and trying to choose the best model for each optimizer to examine differences in prediction: model trained with SGD optimizer for 150 epochs with lr=0.01, model trained with Adam optimizer for 100 epochs with lr=0.01, and model trained with AdamW optimizer for 100 epochs with lr=0.01.
Fig. 6.1 shows the mAP50-95(B)'s evolution during the three models' training across the epochs: while Adam and AdamW's mAP function keeps increasing, SGD remains relatively constant. This is mainly due to the fact that the weight initialization for the 150 SGD 0.01 model was already good, causing the network to get stuck in an optimal state.

The **prediction on unseen data** was not particularly meaningful since all three models performed quite well.

It was the **laboratory testing** that ultimately guided the decision in favor of the SGD model. During the laboratory experiments, it was the only model capable of correctly segmenting the apple object. Both Adam and AdamW produced inaccurate masks and failed to segment when the object was too close to the camera. The mask size continuously fluctuated, even when the mock-up apple was stationary.

IS model	precision(B)	recall(B)	train/mAP50-95(B)	train/mAP50(B)	val/mAP50-95	val/mAP50
100, Adam, 0.01	0.84013	0.7998	0.68332	0.87258	0.68419	0.8729
100, Adam, 0.001	0.8444	0.82124	0.6936	0.87543	0.69403	0.87551
100, Adam, 0.0001	0.85994	0.80334	0.68001	0.87179	0.6821	0.8693
100, AdamW, 0.01	0.84816	0.81416	0.68751	0.87755	0.68838	0.8778
100, AdamW, 0.001	0.85746	0.81337	0.6961	0.86882	0.6978	0.8689
100, AdamW, 0.0001	0.85757	0.81141	0.68216	0.86901	0.6817	0.8641
100, SGD, 0.01	0.86115	0.80649	0.69307	0.86526	0.69477	0.8655
100, SGD, 0.001	0.86033	0.80098	0.68418	0.86754	0.68281	0.8653
150, Adam, 0.01	0.84919	0.81101	0.68421	0.87469	0.68546	0.8740
150, Adam, 0.001	0.85449	0.81377	0.69091	0.87708	0.69011	0.8748
150, Adam, 0.0001	0.84552	0.80924	0.68959	0.87812	0.69084	0.8764
150, AdamW, 0.01	0.85536	0.81455	0.68962	0.87541	0.69129	0.8758
150, AdamW, 0.001	0.85577	0.80913	0.68902	0.87437	0.69026	0.8745
150, AdamW, 0.0001	0.84481	0.80964	0.68871	0.8773	0.68967	0.8771
150, SGD, 0.01	**0.85568**	**0.81023**	**0.68649**	**0.8723**	**0.6861**	**0.8696**
150, SGD, 0.001	0.84683	0.81141	0.68955	0.87777	0.69023	0.8768
150, SGD, 0.0001	0.84662	0.81062	0.68954	0.87834	0.68966	0.8776
200, Adam, 0.01	0.85983	0.80806	0.68741	0.87135	0.68735	0.8696
200, Adam, 0.001	0.86206	0.81278	0.69088	0.87725	0.69196	0.8739
200, Adam, 0.0001	0.85667	0.80477	0.68872	0.87558	0.68999	0.8741
200, AdamW, 0.01	0.86376	0.80688	0.68696	0.86833	0.68795	0.8661
200, AdamW, 0.001	0.86039	0.81474	0.69089	0.87574	0.69321	0.8715
200, AdamW, 0.0001	0.85562	0.79997	0.68822	0.87513	0.68999	0.8741
200, SGD, 0.01	0.86316	0.80452	0.68799	0.87052	0.68953	0.8683
200, SGD, 0.001	0.86027	0.81065	0.6895	0.87488	0.68985	0.8735
200, SGD, 0.0001	0.85968	0.80551	0.6874	0.87575	0.68884	0.8748

Table 6.1: Results obtained with Grid Search training configurations for IS models

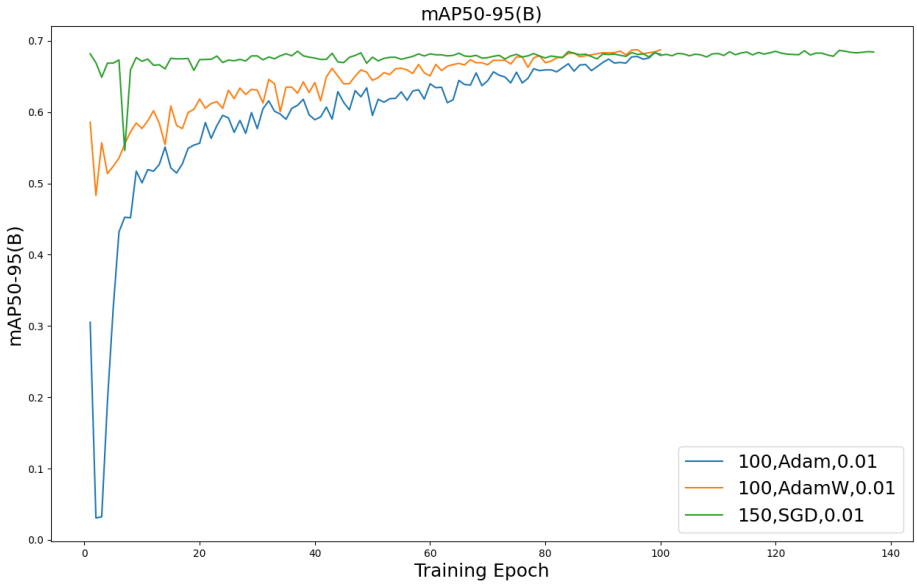


Figure 6.1: Representation of mAP50-95(B)'s evolution across epochs for the three IS selected models

Figure 6.2: Example of inference by 'Adam_100_0.01' model

Figure 6.3: Example of inference by 'SGD_150_0.01' model

Thus, the choice of the YOLOv8 CNN, fine-tuned with SGD optimizer for 150 epochs with 0.01 as learning rate, was made to ensure stable feature extraction for feeding the IBVS controller.

The Fig. 6.4 represents the results provided by YOLO after the training of

the chosen CNN: the loss values for various components (Box, Segmentation, Classification and Distributed Focal Loss) generally have a decreasing trend over the epochs for both train and validation sets. Even if some fluctuations are visible (e.g. train/seg_loss), the tiny range of variations indicates that the function remains quite constant, due to the good weight initialization of the network. To sum it up, the selected optimal YOLOv8 CNN model for Instance Segmentation task shows:

- train/mAP50(B) = 87.23%;
- val/mAP50 = 86.96%;
- train/mAP50-95(B) = 68.65%;
- val/mAP50-95 = 68.61%.

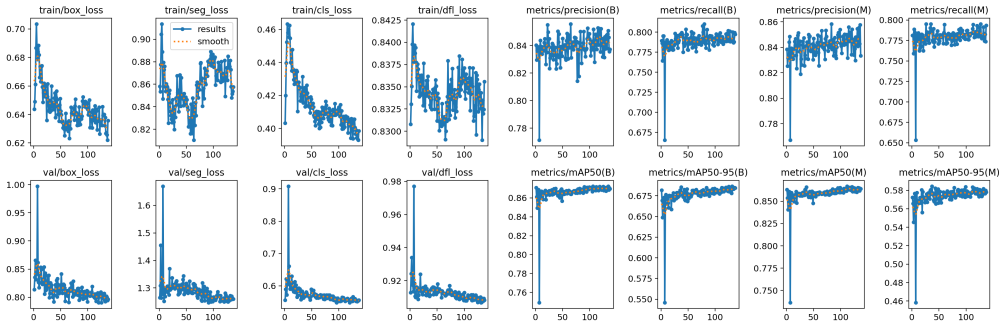


Figure 6.4: Results provided by YOLO of the IS model trained for 150 epochs with SGD optimizer and lr=0.01

6.1.2 Instance Segmentation and Multi-class Classification model results

The aforementioned procedure to select the optimal model among the trained ones was adopted for YOLOv8 CNN models for Instance Segmentation and Multi-class Classification tasks too. Tab. 6.2 shows the results of some of the trained models. As first remark, all models trained on a fine-tuned IS segmentation model, obtained from previous section, got better values in terms of mAP50-95(B), precision(B) and recall(B), with respect to those models trained on YOLOv8s-Seg model. This is the reason why only these kind of models were therefore considered for further analysis. Fig. 6.5 shows the mAP50-95(B)'s evolution across the training epochs for the four models that were selected after the **initial analysis**: 'PT_Adam_Adam_200_0.01' (model trained

with Adam optimizer for 200 epochs with lr=0.01, using 100,Adam,0.01 as pre-trained model); 'PT_SGD_Adam_200_0.01' (model trained with Adam optimizer for 200 epochs with lr=0.01, using 100,SGD,0.01 as pre-trained model); 'PT_SGD_SGD_200_0.01' (model trained with SGD optimizer for 200 epochs with lr=0.01, using 100,SGD,0.01 as pre-trained model); 'PT_SGD_AdamW_200_0.01' (model trained with AdamW optimizer for 200 epochs with lr=0.01, using 100,SGD,0.01 as pre-trained model). Let's notice from Fig. 6.5 that the mAP50-95(B) function for 'PT_SGD_SGD_200_0.01' model stops epochs before than 200th: indeed, to avoid overfitting, YOLOv8 early ends the training if, after 100 epochs of waiting, it doesn't see any improvement in validation metrics.

IS and MC	**Pre-trained**	**precision**	**recall**	**train**		**val**	
model	**model**	**(B)**	**(B)**	**mAP50-95(B)**	**mAP50(B)**	**mAP50-95**	**mAP50**
200 Adam 0.01	100 Adam 0.01	0.5986	0.67026	0.49961	0.63121	0.5001	0.6312
200 Adam 0.01	100 SGD 0.01	0.61099	0.6572	0.50009	0.63611	0.5015	0.6359
200 AdamW 0.01	100 SGD 0.01	0.61778	0.65334	0.48578	0.61046	0.4870	0.6090
200 SGD 0.01	**100 SGD 0.01**	**0.6107**	**0.67301**	**0.51234**	**0.6389**	**0.5135**	**0.6394**
100 Adam 0.01	YOLOv8s-Seg	0.57396	0.63483	0.47763	0.60288	0.4793	0.60355
200 SGD 0.01	YOLOv8s-Seg	0.59502	0.63325	0.48035	0.60587	0.48117	0.60519
100 AdamW 0.01	100 Adam 0.01	0.61206	0.66697	0.50123	0.63591	0.5129	0.6367
130 AdamW 0.01	100 Adam 0.01	0.60239	0.67233	0.49899	0.6286	0.50077	0.6279

Table 6.2: Results obtained with various training configurations for IS and MC models

The **prediction on unseen data** phase reported that the best models in prediction resulted: 'PT_Adam_Adam_200_0.01', 'PT_SGD_Adam_200_0.01' and 'PT_SGD_SGD_200_0.01'.

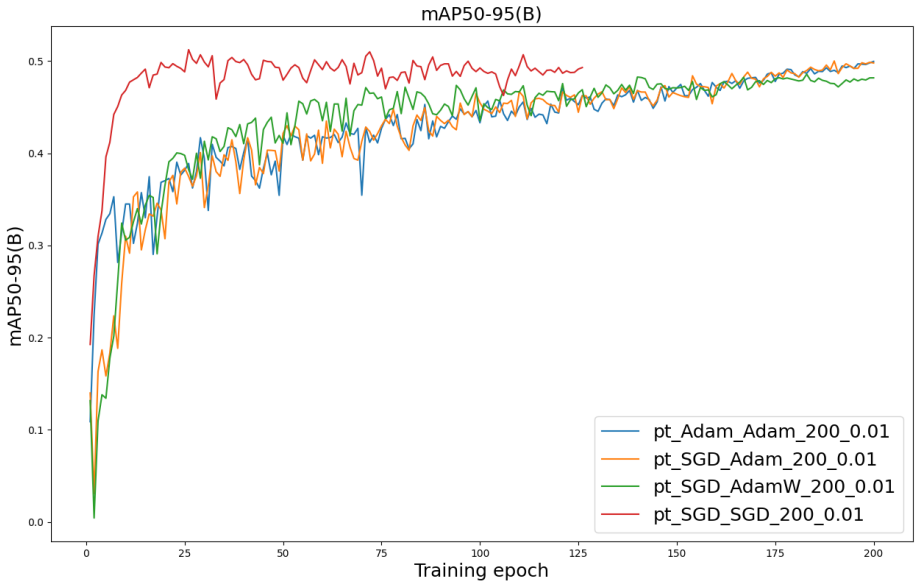


Figure 6.5: Representation of mAP50-95(B)'s evolution across epochs for the IS and MC selected models

Figure 6.6: Example of prediction by 'PT_SGD_Adam_200_0.01' model

Figure 6.7: Example of prediction by 'PT_Adam_Adam_200_0.01' model

Figure 6.8: Example of prediction by 'PT_SGD_SGD_200_0.01' model

At **laboratory testing** with the mock-up apple, the best model at inferencing was 'PT_SGD_SGD_200_0.01' since it captured the entire mask of the apple even upon proximity to it. This factor led to the selection of 'PT_SGD_SGD_200_0.01' as the optimal model configuration for Multi-class Classification and Instance Segmentation tasks.
Nevertheless, an issue arose both in prediction of unseen videos and in the inference tests for all MC models, namely the instability of the network to classify the apple instances frame by frame: the class label assigned to the apple object changed too quickly.

Figure 6.9: Example of inference by 'PT_SGD_Adam_200_0.01' model

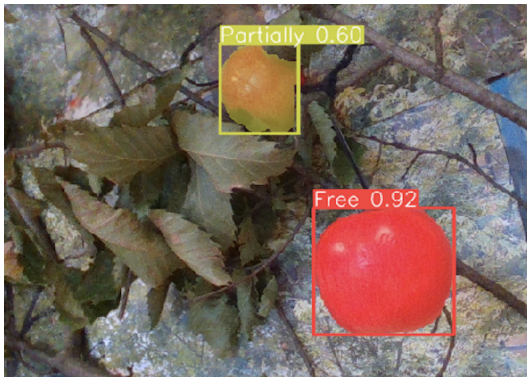


Figure 6.10: Example of inference by 'PT_SGD_SGD_200_0.01' model

Fig. 6.11 represents the results provided by YOLO after the training of the chosen CNN, fine-tuned on the pre-trained IS model (100,SGD,0.01) for 200 epochs with

SGD optimizer and using 0.01 as learning rate.

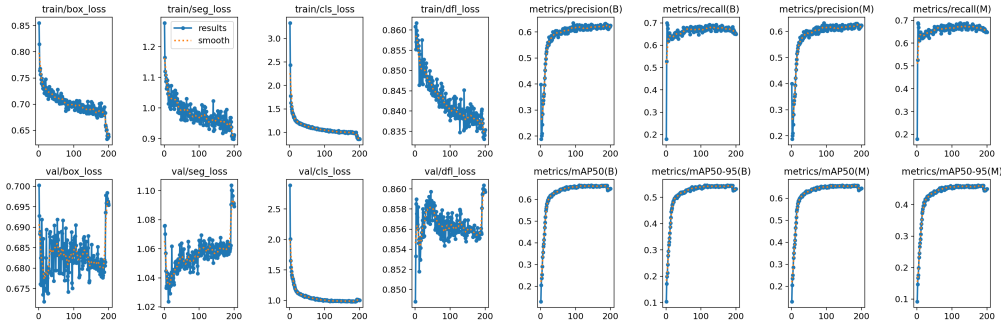


Figure 6.11: Results of the IS and MC model trained for 200 epochs with SGD optimizer and lr=0.01, on the pre-trained 100,SGD,0.01 IS model

Finally, to conclude, the selected optimal YOLOv8 CNN model for Multi Classification and Instance Segmentation tasks features:

- train/mAP50(B) = 63.89%;
- val/mAP50 = 63.94%;
- train/mAP50-95(B) = 51.23%;
- val/mAP50-95 = 51.35%.

6.2 Experimental tests on IBVS and overall system architecture

Extensive experiments were conducted on the overall system architecture, modeled as a Finite State Machine (FSM) (Fig. 4.5), to evaluate the individual performances of the Vision and Controller modules and their combined effectiveness.

Experimental setup

A description of the experimental setup follows:

- Kinova Gen3 Lite 6 DoF: operating from a stationary base with static targets;
- Intel RealSense D435i: mounted on the Kinova's end effector in an eye-in-hand camera configuration;
- developed ROS2 software architecture: executed on an Intel NUC11PAH-01, powered by an Intel Core i5-1135G7 CPU;

- the average inference speed of the YOLOv8 CNN, accelerated by Openvino: 5 fps, which reduces the RealSense frame rate (30 fps) and causes the entire architecture to operate at 5 Hz;
- as illustrated in 6.12, a realistic scene was recreated at PIC4SeR laboratory using a plant with leaves, branches, and a lightweight mockup apple, visible from the robot's perspective and within its configuration space, to be consistent with the dataset on which the vision model was trained.

Figure 6.12: Experimental setup for the tests on the IBVS and overall architecture

Experimental tests

Let's refer to the term *scenario* as a specific robot joint configuration and a particular apple position in the scene.
Two types of tests were conducted.

The first set of tests, referred to as **Fixed Initial Configuration Experiments**, aimed to quantitatively evaluate the metrics described below in controlled conditions. Consequently, the scenario comprised a *fixed starting joint configuration*, that

we can call 'home' pose, and a *variable apple position in the scene* for each test. A total of five tests were performed, each consisting of five attempts under the same conditions (i.e., 'home' robot pose and same apple position in the scene) to ensure repeatability. Test 1 was repeated twice, using both the Instance Segmentation CNN and Multi-class Classification model, whereas Test 2,3,4,5 employed only the Instance Segmentation model, which demonstrated superior performance during Test 1.

The second set of tests, referred to as **Variable Initial Configuration Experiments**, aimed to assess the system's robustness and adaptability. A total of seven tests were conducted, each with five attempts. In this second scenario, both the initial robot joint configuration and the apple position varied across tests. This variability allows us to examine how the system performs when starting from different configurations and how the metrics, such as the Closed Loop Time, Harvest Time, and Successful Picking Rate, fluctuate under these dynamic conditions, providing valuable insights into the system's flexibility and operational efficiency in different scenarios.

For each attempt of each test, the FSM was run.
The success of each state of the FSM was evaluated as follows:

- STATE: `Closed Loop` = SUCCESS only if the IBVS correctly guides the end effector toward the target;
- STATE: `Open Loop Grasping` = SUCCESS only if the gripper successfully picks the apple;
- STATE: `Place` = SUCCESS if the motion plan to place the apple occurred without dropping it or encountering singularity.

The evaluation metrics

The metrics used to evaluate the performances of the proposed system for apple harvesting include:

- **Closed Loop Time** (CLT) [s], representing the duration from `Start Controller` state to the end of `Closed Loop` (CL) state. The mean and standard deviation of the CLT were calculated across all successful attempts in each test, where the CL state ended as expected, with the end effector toward the target:

$$\mu_{CLT} = \frac{1}{N} \sum_{i=1}^{N} t_i$$

$$\sigma_{CLT} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(t_i - \mu)^2}$$

where: t_i is the CLT for the i-th successful attempt, N is the total number of successful attempts where the closed loop state was finished as expected, μ_{CLT} is the Mean CLT, σ_{CLT} is the Standard Deviation of CLT.

- **Euclidean Norm of the Error for Each Feature at Steady State**: The Euclidean norm of the error was computed separately for each feature s_i's projected normalized coordinates in the image plane (Eq. 4.7) at steady state, that is, at the end of the closed-loop phase. For each test, the norm for the i-th feature was calculated for each of the five attempts, and then, the mean and standard deviation of the error norm were then computed across the five attempts:

$$|e(\infty)|_{s_i} = \sqrt{e_{x_i}^2 + e_{y_i}^2}$$

where $e_{x_i} = x_i(CLT) - x_i^*$ and $e_{y_i} = y_i(CLT) - y_i^*$ are the x and y error components computed between the normalized projected coordinates in the image plane of the i-th current feature and the i-th desired feature, at steady state. For each feature s_i, the mean $\mu_{|e(\infty)|_{s_i}}$ and standard deviation $\sigma_{|e(\infty)|_{s_i}}$ of the error norm across all attempts were calculated as:

$$\mu_{|e(\infty)|_{s_i}} = \frac{1}{5}\sum_{k=1}^{5}|e(\infty)|_{s_i}^{(k)}$$

$$\sigma_{|e(\infty)|_{s_i}} = \sqrt{\frac{1}{5}\sum_{k=1}^{5}\left(|e(\infty)|_{s_i}^{(k)} - \mu_{|e(\infty)|_{s_i}}\right)^2}$$

where $|e(\infty)|_{s_i}^{(k)}$ is the error norm for the i-th feature during the k-th attempt, and 5 is the number of attempts for each test.

- **Harvest Time** (HT) [s], defined as the duration from `Start Controller` state to the end of `Place` state. The mean and standard deviation of the HT were calculated across all successful attempts in each test, where the `Place` state was ended as expected, without the occurrence of an early apple drop or robot singularity:

$$\mu_h = \frac{1}{N}\sum_{i=1}^{N}h_i$$

$$\sigma_h = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(h_i - \mu_h)^2}$$

where: h_i is the harvest time (from `Start Controller` to `Place` state) for the i-th successful attempt, N is the total number of successful attempts where the `Place` state was correctly finished, μ_h is the Mean HT, σ_h is the standard deviation of the HT.

- **Successful Picking Rate**: This metric is calculated as the ratio of the number of successful attempts (where the process correctly finishes the `Place` state from `Start Controller`) out of the total number of attempts:

$$\text{Successful Picking Rate} = \frac{N_{\text{successful}}}{N_{\text{total}}}$$

where $N_{\text{successful}}$ is the number of successful attempts, N_{total} is the total number of attempts.

Tuning Parameters

All experiments were conducted with the following parameter settings:

- λ=1, proportional gain scalar in IBVS control law, playing a role in the convergence speed, adjusted to optimize the performance;
- f_{IBVS}=20Hz, frequency of the IBVS controller, set to more than twice the 5 Hz vision update rate, to guarantee both stability and responsiveness;
- $|\mathbf{v}_{tf}|_{\text{saturated}} = 0.06\,\text{m/s}$, to minimize excessive oscillations;
- α_{EMA}= 0.8, to ensure a quicker response while smoothing the output as well;
- Δ_{OL}=2.4s, to prevent abrupt accelerations that could excessively push the lightweight apple;
- $w_a = 0.4$, $w_c = 0.6$: tunable weights of the cost function, chosen to prioritize the object's area in order to mitigate the frame by frame classification instability observed in the Multi-class Classification network;
- the target apple was positioned between 36 cm and 78 cm along the z-axis of the Kinova tool frame across tests.

6.2.1 Results of Fixed Initial Configuration Experiments

These tests were conducted with a Fixed Initial joint Configuration (q_{home}), called *home*, varying the location of the apple in the scene (resulting in the object seen in the center of the image, top left, top right, down left, down right).

$$q_{home} = \begin{bmatrix} -89^\circ & -22^\circ & 67^\circ & -1^\circ & -116^\circ & 0^\circ \end{bmatrix}$$

The *home* starting position was chosen to allow the Kinova robot to maneuver effectively in multiple directions, avoiding singularities.

In the following, the detailed setup and schematic results about the Fixed Initial Configuration Experiments will be provided.

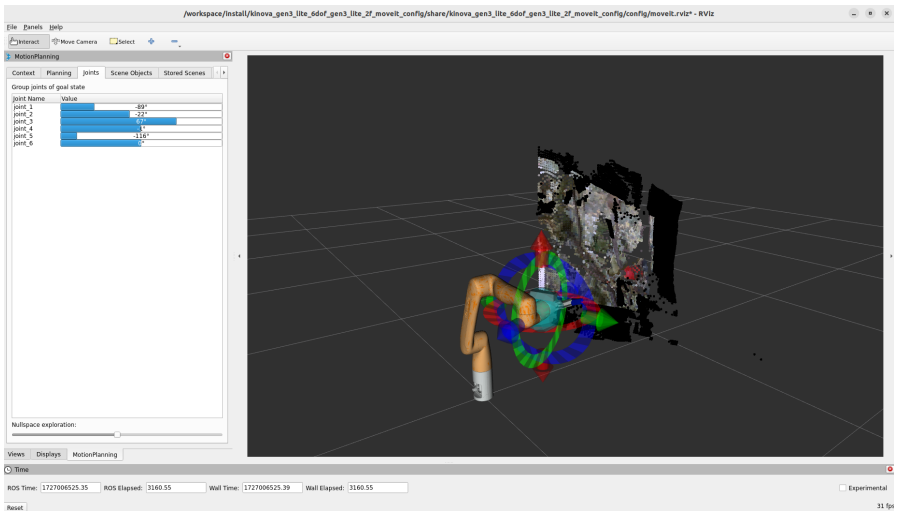


Figure 6.13: View of the Kinova's *home* position from MoveIt 2

TEST 1

CNN model: Instance Segmentation (IS) model
View from RealSense:

Figure 6.14: View from RealSense for Test n. 1 and selected target with IS model

Tab. 6.3 shows the results obtained for each attempt of Test 1. Notice that: A refers to Attempt, OL_G refers to Open Loop Grasping, CLT refers to Closed Loop Time, HT refers to Harvest Time, NV stands for 'not valid'.

Test 1 IS	CL		OL_G	PLACE	
	Success	**CLT**	**Success**	**Success**	**HT**
A1	True	10.78 s	True	True	21.20 s
A2	True	11.00 s	True	True	21.41 s
A3	True	10.78 s	True	True	21.17 s
A4	True	10.70 s	True	True	21.02 s
A5	True	10.37 s	True	True	20.71 s

Table 6.3: Results of the five attempts for Test n. 1 with IS model

The following graphs illustrate:

- the evolution of the error components on the pixel coordinates u and v for the five visual features over time computed between the desired constant features and the current features extracted by the vision module, from `Start Controller` state to the end of `Closed Loop` phase, providing insights into the controller's behavior and effectiveness;

- the linear and angular components of the Cartesian velocity over time, output of the IBVS controller, sent to the robot during the `Closed Loop` phase.

Each plot includes data from the five separate attempts of the test, allowing for the assessment of consistency and robustness across multiple trials of the overall architecture in the same conditions.

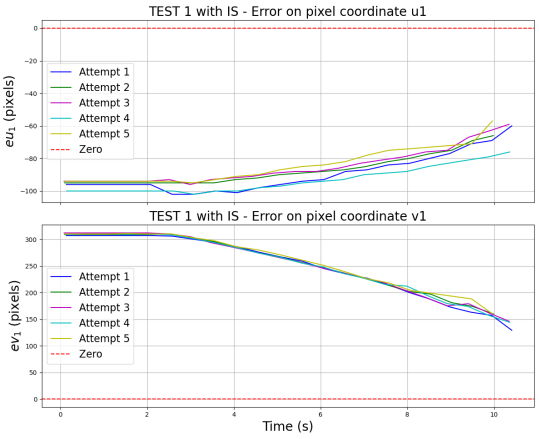


Figure 6.15: Error on feature s1 during closed loop phase for TEST1 with IS model

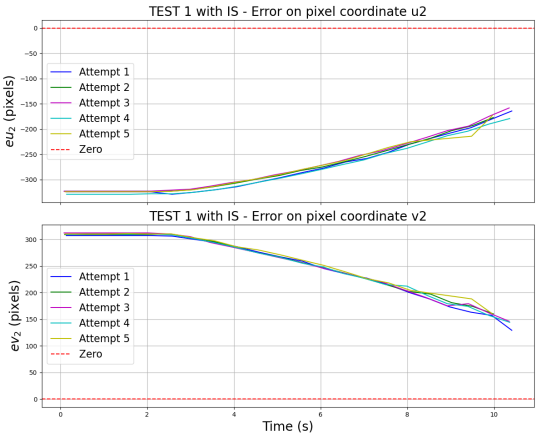


Figure 6.16: Error on feature s2 during closed loop phase for TEST1 with IS model

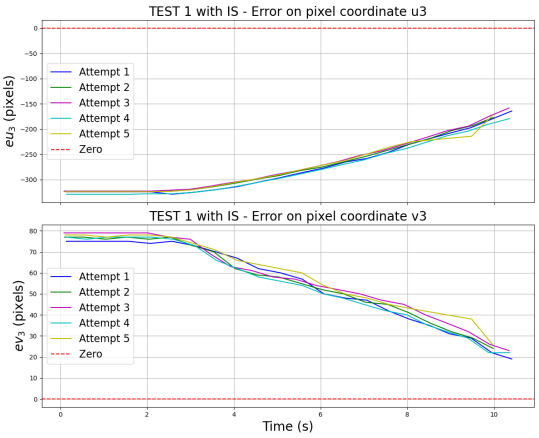


Figure 6.17: Error on feature s3 during closed loop phase for TEST1 with IS model

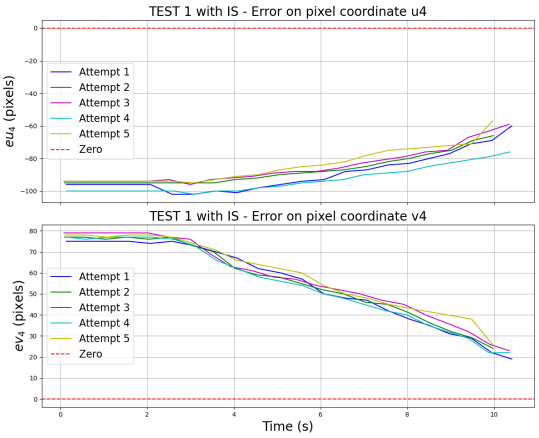


Figure 6.18: Error on feature s4 during closed loop phase for TEST1 with IS model

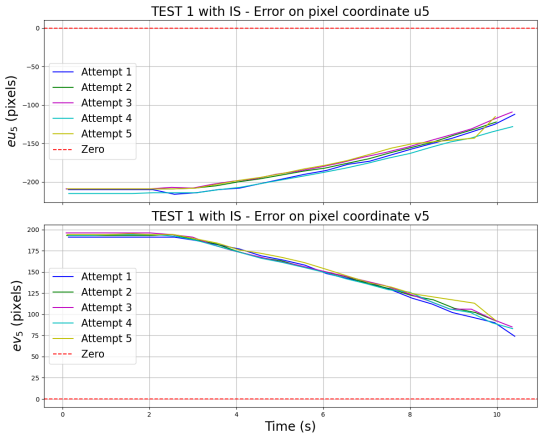


Figure 6.19: Error on feature s5 during closed loop phase for TEST1 with IS model

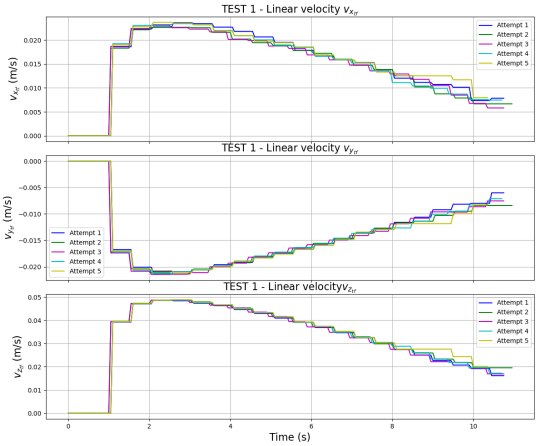


Figure 6.20: Linear velocity during closed loop phase for TEST1 with IS model

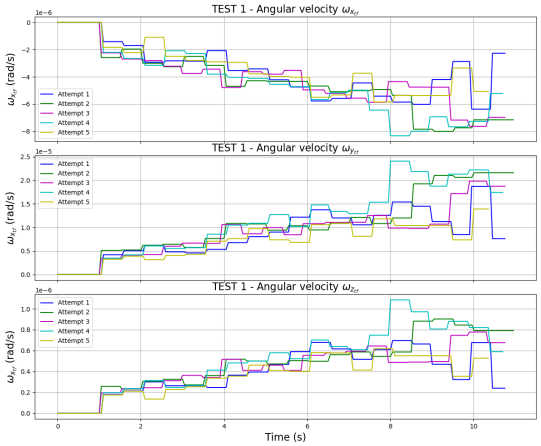


Figure 6.21: Angular velocity during closed loop phase for TEST1 with IS model

It turns out that results of Test 1 achieved with IS model were positive, resulting in 5 successful attempts out of 5 total. The Instance Segmentation is perfectly working, feeding the IBVS controller with coherent feature coordinates extraction, leading to a general error convergence of the features coordinates toward zero over time. There's consistency across the five attempts, with minor variations likely due to slight differences in initial conditions or environmental factors. However, step-like changes can be observed both in Error evolution and in the Cartesian Velocity Commands: this is mainly due to the digital implementation of the discrete control

system and in the relatively slow inference speed of the Instance Segmentation model with respect to control frequency. Anyway, the Successful Picking Rate (5/5) across the attempts of Test1 suggests that both the IS model and IBVS controller worked as intended.

CNN model: Instance Segmentation (IS) and Multi-class Classification (MC) model
View from RealSense:

Figure 6.22: View from RealSense for Test n. 1 and selected target with IS and MC model

Test 1 MC	**CL**		**OL_G**	**PLACE**	
	Success	**CLT**	**Success**	**Success**	**HT**
A1	True	8.80 s	True	True	19.18 s
A2	False	NV	False	False	NV
A3	False	NV s	False	False	NV
A4	True	8.97 s	True	True	19.46 s
A5	False	NV	False	False	NV

Table 6.4: Results of the five attempts for Test n. 1 with MC and IS model

A2 failed the CL phase because the network firstly detected the target seen from Fig. 6.22 and suddenly switched to the second (hidden) apple as target. After

that, the robot entered the open loop without reaching the real target apple, since it found the branch as obstacle ahead. A3 and A4 failed because the MC model continuously changed the target, leading to the robot being stopped in the wrong final pose.

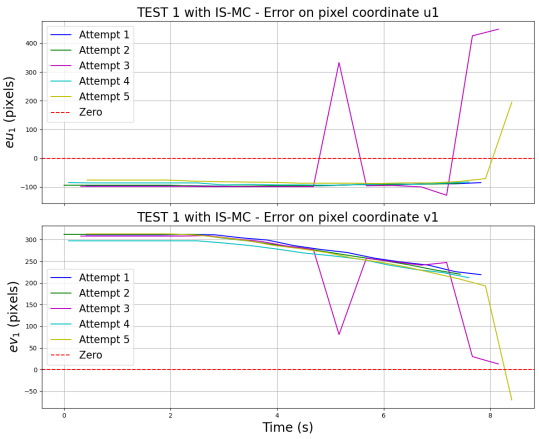


Figure 6.23: Error on feature s1 during closed loop phase for TEST1 with IS MC model

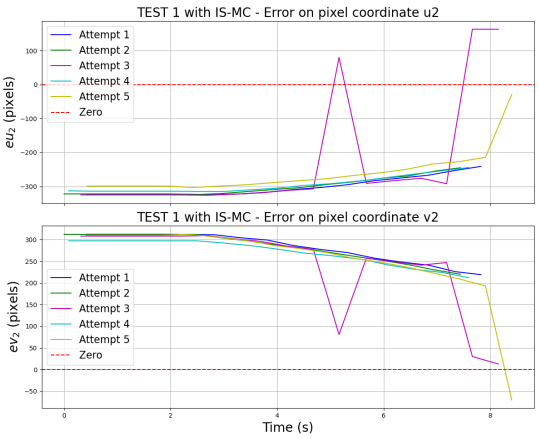


Figure 6.24: Error on feature s2 during closed loop phase for TEST1 with IS MC model

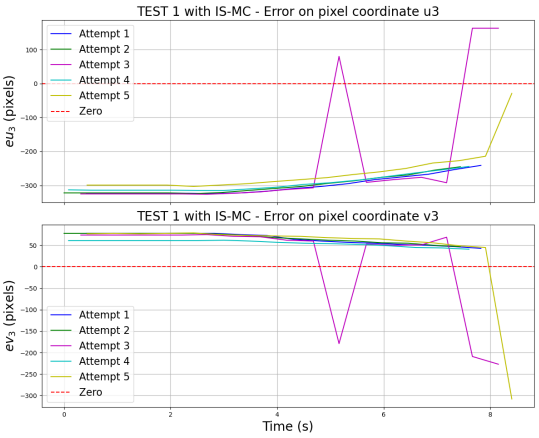


Figure 6.25: Error on feature s3 during closed loop phase for TEST1 with IS MC model

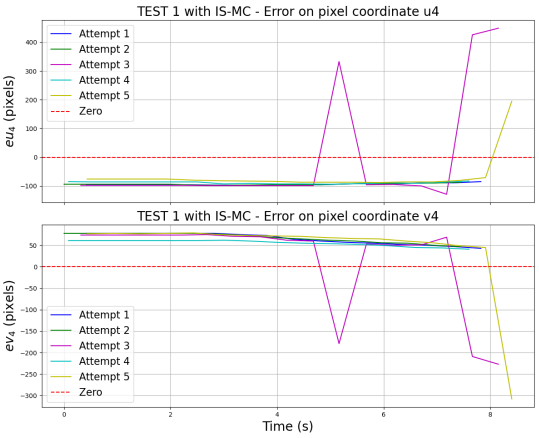


Figure 6.26: Error on feature s4 during closed loop phase for TEST1 with IS MC model

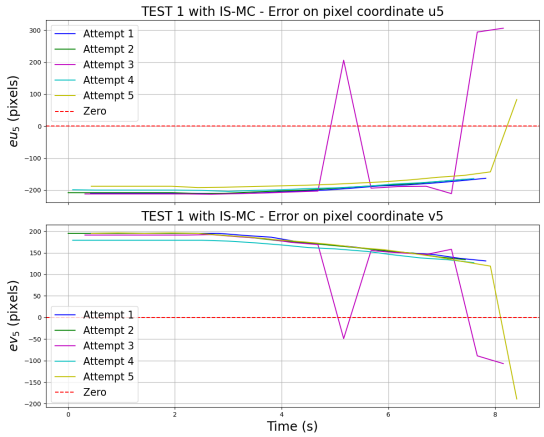


Figure 6.27: Error on feature s5 during closed loop phase for TEST1 with IS MC model

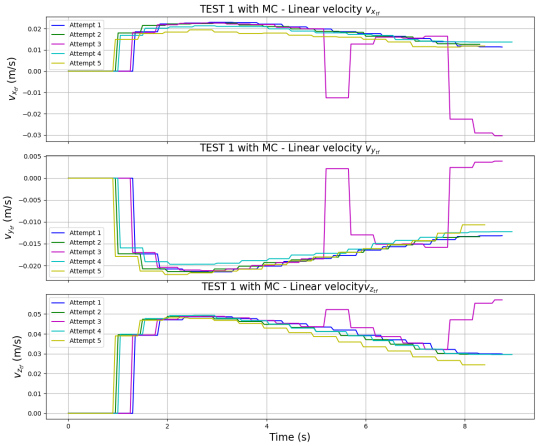


Figure 6.28: Linear velocity during closed loop phase for TEST1 with IS MC model

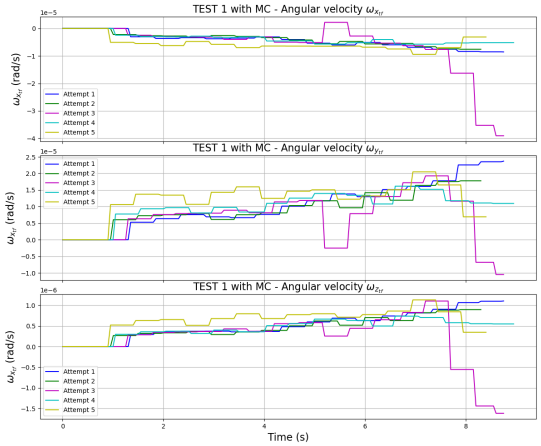


Figure 6.29: Angular velocity during closed loop phase for TEST1 with IS MC model

Then, the results of Test 1 with IS and MC model were unsatisfactory, yielding a lower Successful Picking Rate (2/5) across the attempts of Test1, compared to the previous case. Although the IS and MC network is able to segment the instances well, it tends to change the class-label of the instances too quickly. This rapid change led to variable targets in the cost function, compromising stability. The extraction of the feature coordinates is thus unstable: in the error and velocity plots for A3 and A5, noticeable spikes occurred due to the shifting target apple's position, resulting in the divergence of the error.

In conclusion, the network for *Instance Segmentation* only was selected as Vision Model to perform inference on the RGB data for the following tests.

TEST 2

CNN model: Instance Segmentation model
View from RealSense:

Figure 6.30: View from RealSense for Test n. 2

Test 2	**CL**		**OL_G**	**PLACE**	
	Success	**CLT**	**Success**	**Success**	**HT**
A1	True	8.47 s	True	True	18.99 s
A2	True	9.00 s	False	False	NV
A3	True	8.98 s	True	True	19.41 s
A4	True	8.98 s	True	True	19.98 s
A5	True	9.00 s	True	False	NV

Table 6.5: Results of the five attempts for Test n. 2

In A2, the grasping attempt was unsuccessful due to the end effector's excessive forward motion, which, combined with the apple's minimal weight, caused the object to move excessively during the picking. In A5, the same situation of A2

occurred. Initially, the apple was grasped but it slipped from the gripper and fell before the robot could transition to the placement phase.

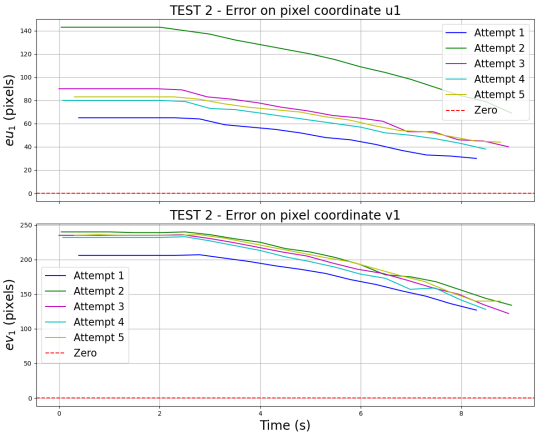


Figure 6.31: Error on feature s1 during closed loop phase for TEST2 with IS model

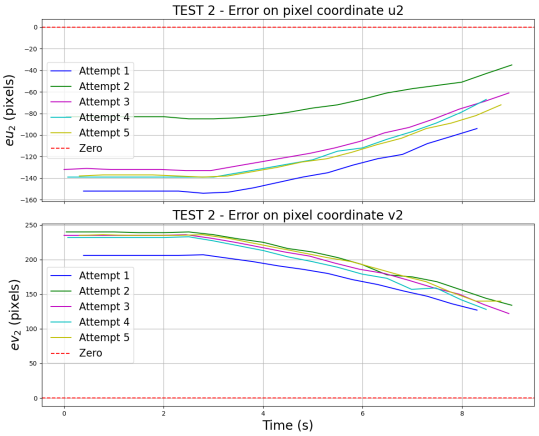


Figure 6.32: Error on feature s2 during closed loop phase for TEST2 with IS model

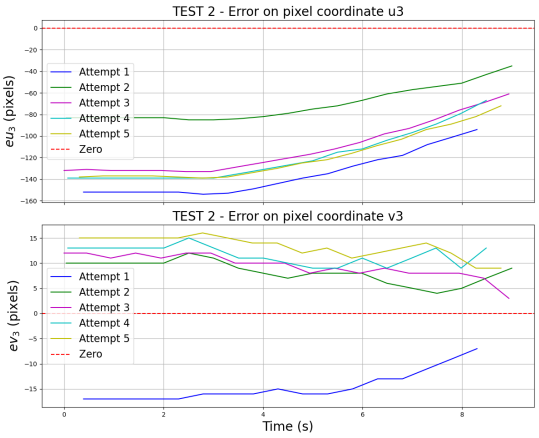


Figure 6.33: Error on feature s3 during closed loop phase for TEST2 with IS model

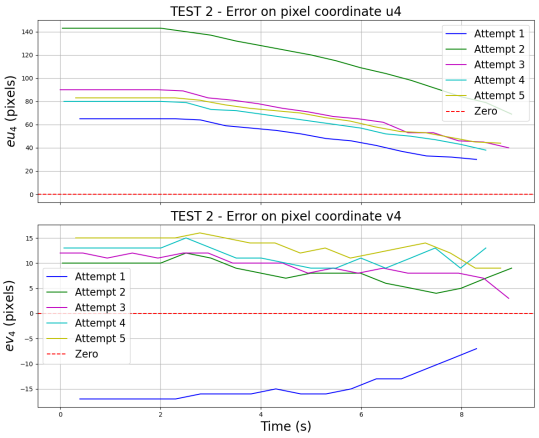


Figure 6.34: Error on feature s4 during closed loop phase for TEST2 with IS model

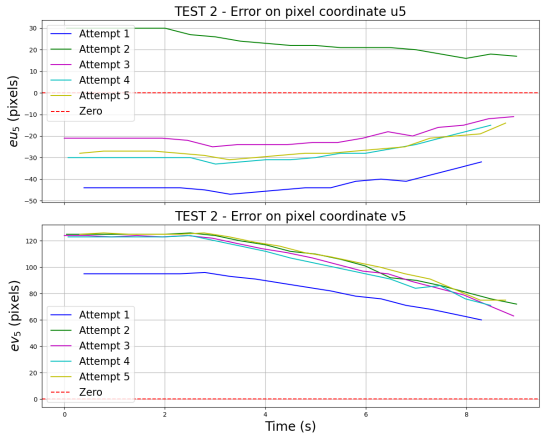


Figure 6.35: Error on feature s5 during closed loop phase for TEST2 with IS model

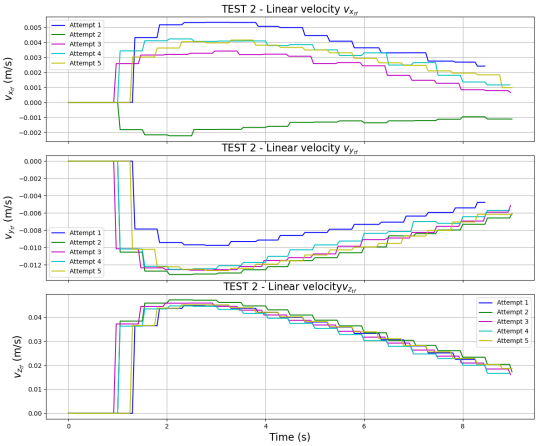


Figure 6.36: Linear velocity during closed loop phase for TEST2 with IS model

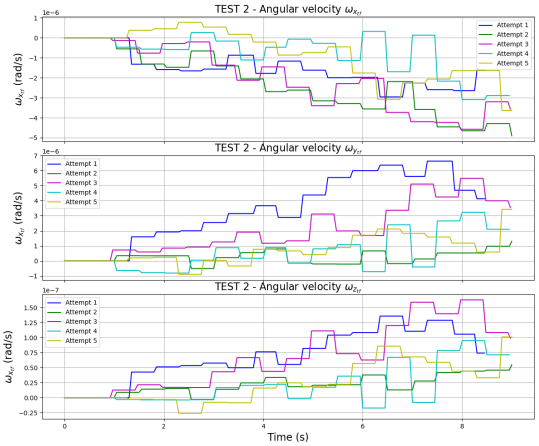


Figure 6.37: Angular velocity during closed loop phase for TEST2 with IS model

Based on the plots from Test 2, we can observe some variations in the initial error of certain features parameters across the different attempts. This discrepancy can be attributed to the human error in positioning the apple exactly in the same location for all five trials. Despite these initial differences, it's important to note that the error consistently demonstrates a decreasing trend, converging toward zero in all cases. This convergence indicates that regardless of small initial positioning inconsistencies, the visual servoing system effectively guides the robot towards the desired configuration. All velocities tend towards zero by the end of the execution,

suggesting the robot is slowing down as soon as the current features overlap with the desired features.

TEST 3

CNN model: Instance Segmentation model
View from RealSense:

Figure 6.38: View from RealSense for Test n. 3

Test 3	**CL**		**OL_G**	**PLACE**	
	Success	**CLT**	**Success**	**Success**	**HT**
A1	True	6.50 s	True	True	17.10 s
A2	True	7.00 s	True	False	NV
A3	True	6.49 s	True	True	16.99 s
A4	True	6.64 s	True	True	17.21 s
A5	True	6.00 s	True	False	NV

Table 6.6: Results of the five attempts for Test n. 3

The position of the apple in Test 3 was a bit crucial, due to the vicinity to the branches. Even though a light collision occurred during A1, it didn't have an impact on the tasks completion. On the contrary, in A2, the collision prevented

the Kinova to place the apple, being blocked by a Safety Failure mechanism. In A5, the robot failed to exit from the Open Loop Grasping because it stretched too much to pick the apple and then, from that position, the MoveIt planner wasn't able to find the motion plan to reach the placement pose.

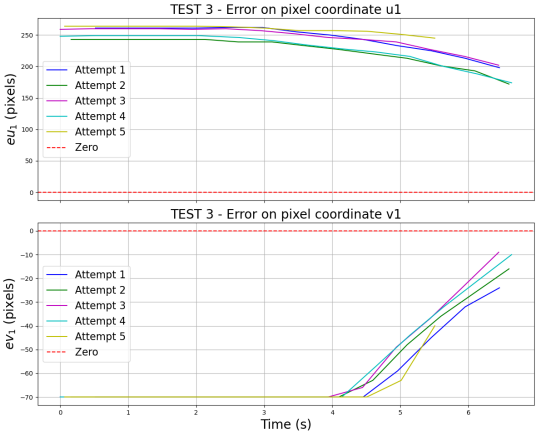


Figure 6.39: Error on feature s1 during closed loop phase for TEST3 with IS model

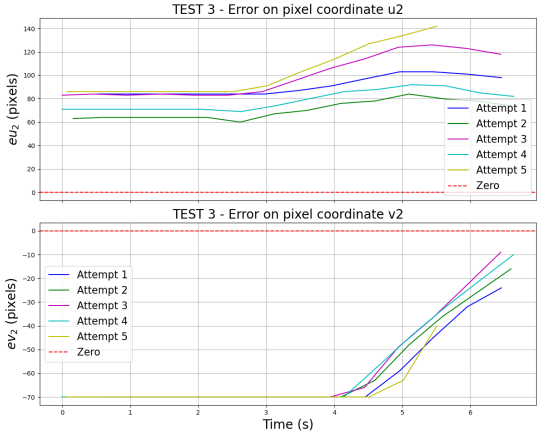


Figure 6.40: Error on feature s2 during closed loop phase for TEST3 with IS model

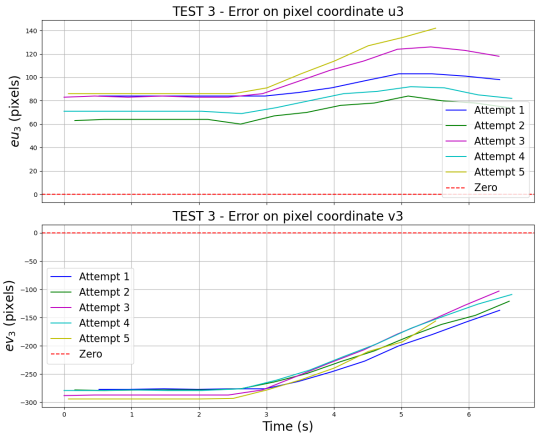


Figure 6.41: Error on feature s3 during closed loop phase for TEST3 with IS model

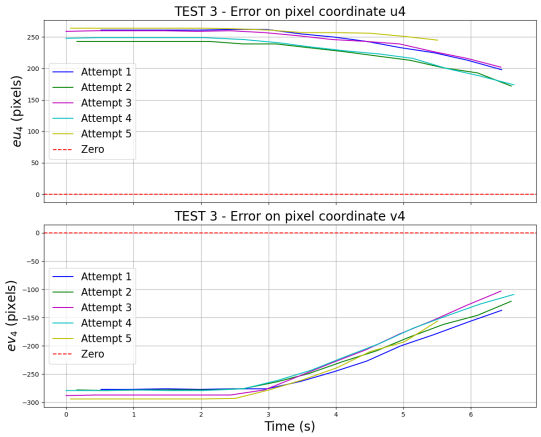


Figure 6.42: Error on feature s4 during closed loop phase for TEST3 with IS model

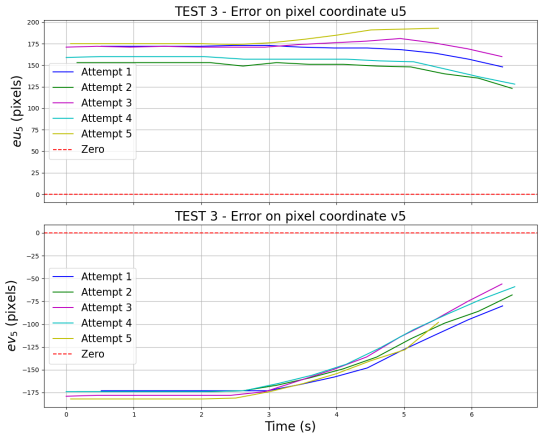


Figure 6.43: Error on feature s5 during closed loop phase for TEST3 with IS model

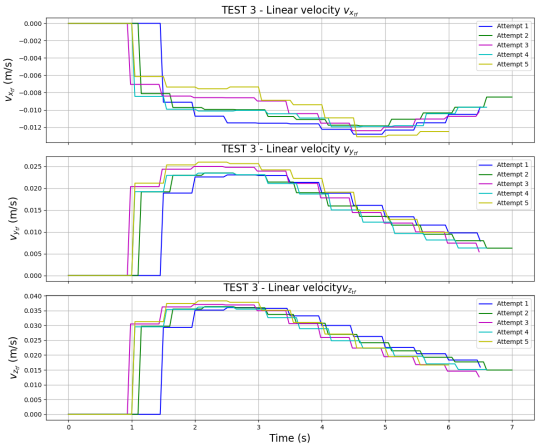


Figure 6.44: Linear velocity during closed loop phase for TEST3 with IS model

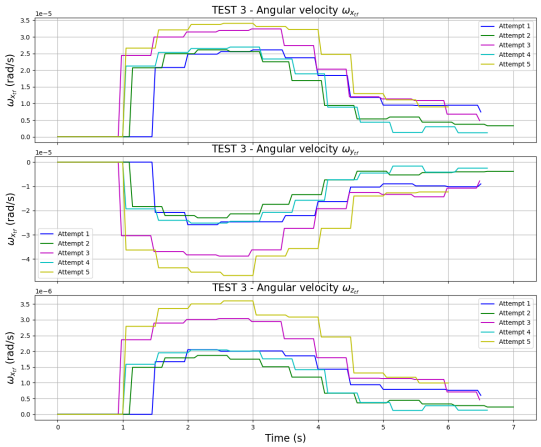


Figure 6.45: Angular velocity during closed loop phase for TEST3 with IS model

The error and velocity plots exhibit notable similarity across all attempts, indicating consistent controller performance. However, A5 stands out due to its shorter closed-loop phase duration and larger velocity command magnitudes. This behavior anticipates the system's entry into the singularity that will block the robot.

TEST 4

CNN model: Instance Segmentation model
View from RealSense:

Figure 6.46: View from RealSense for Test n. 4

Test 4	CL		OL_G	PLACE	
	Success	**CLT**	**Success**	**Success**	**HT**
A1	True	8.07 s	True	True	18.51 s
A2	True	7.97 s	True	True	18.70 s
A3	True	8.00 s	True	True	18.41 s
A4	True	8.04 s	True	False	NV s
A5	True	7.97 s	True	True	18.41

Table 6.7: Results of the five attempts for Test n. 4

In A5, a hard collision blocked the robot that didn't reach the place state.

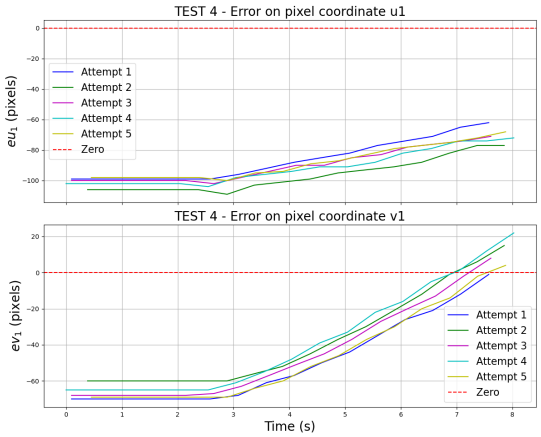


Figure 6.47: Error on feature s1 during closed loop phase for TEST4 with IS model

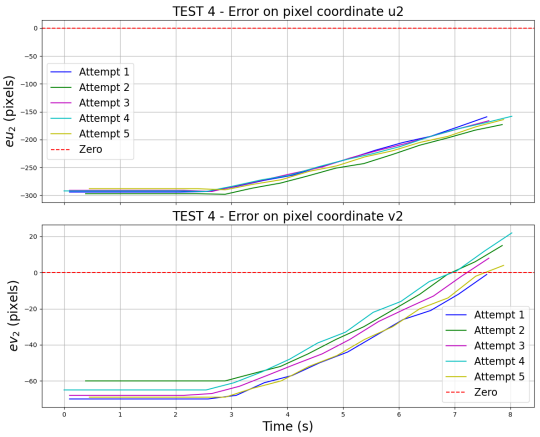


Figure 6.48: Error on feature s2 during closed loop phase for TEST4 with IS model

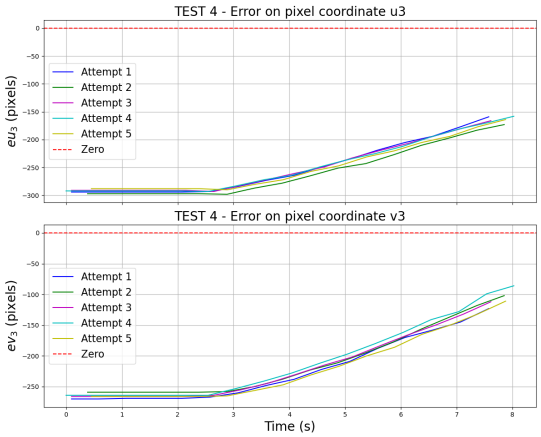


Figure 6.49: Error on feature s3 during closed loop phase for TEST4 with IS model

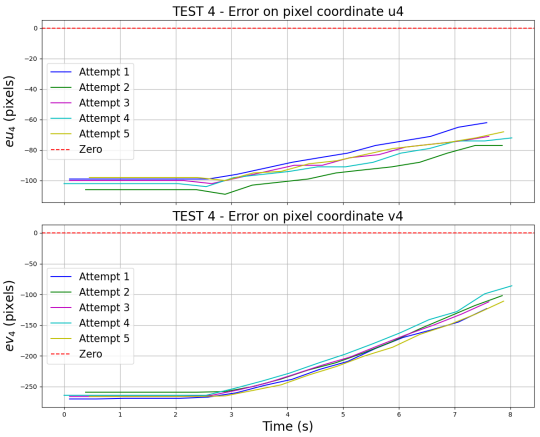


Figure 6.50: Error on feature s4 during closed loop phase for TEST4 with IS model

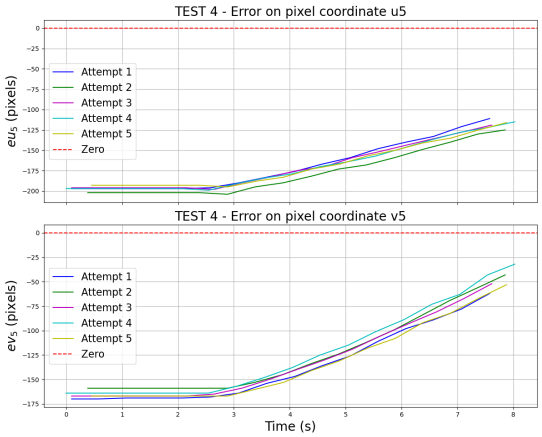


Figure 6.51: Error on feature s5 during closed loop phase for TEST4 with IS model

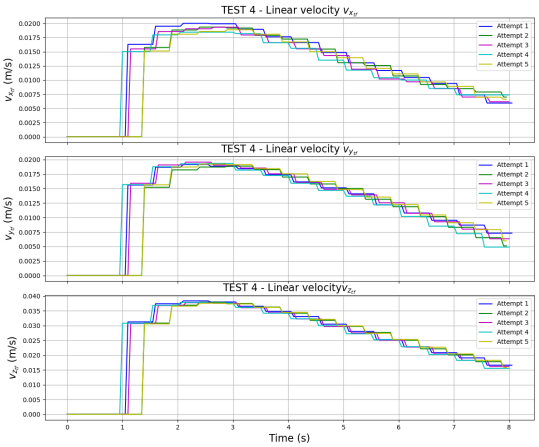


Figure 6.52: Linear velocity during closed loop phase for TEST4 with IS model

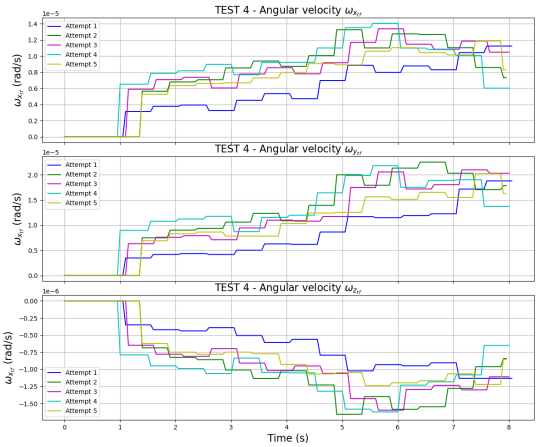


Figure 6.53: Angular velocity during closed loop phase for TEST4 with IS model

Also error and velocity plots of Test 4 show the effectiveness of the controller in achieving error convergence close to zero. The velocity command reduces progressively as the error diminishes, ensuring smooth deceleration as the target is approached.

TEST 5

CNN model: Instance Segmentation model
View from RealSense:

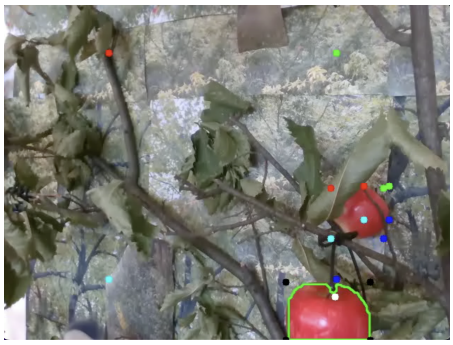


Figure 6.54: View from RealSense for Test n. 5

Test 5	**CL**		**OL_G**	**PLACE**	
	Success	**CLT**	**Success**	**Success**	**HT**
A1	True	7.60 s	True	True	18.30 s
A2	True	8.10 s	True	True	18.74 s
A3	True	7.99 s	True	True	18.69 s
A4	True	7.99 s	True	True	18.60 s
A5	True	8.58 s	True	True	19.10 s

Table 6.8: Results of the five attempts for Test n. 5

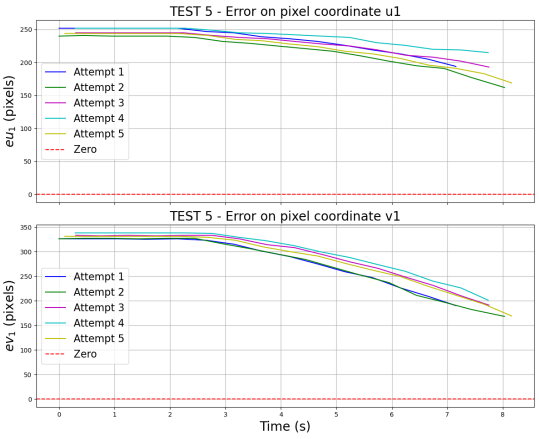


Figure 6.55: Error on feature s1 during closed loop phase for TEST5 with IS model

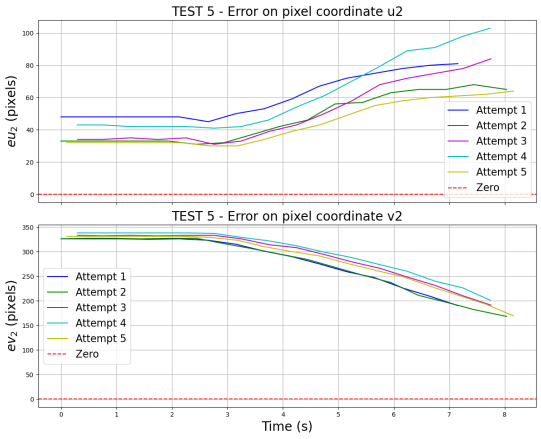


Figure 6.56: Error on feature s2 during closed loop phase for TEST5 with IS model

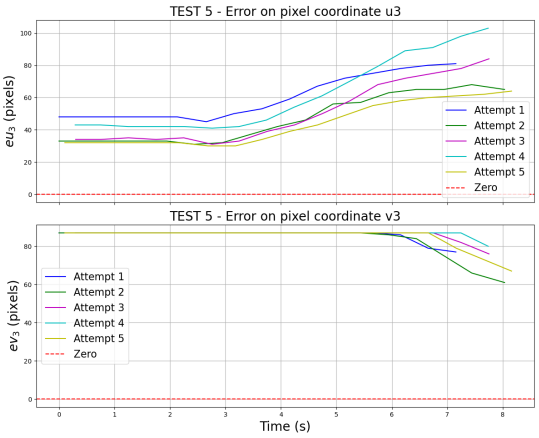


Figure 6.57: Error on feature s3 during closed loop phase for TEST5 with IS model

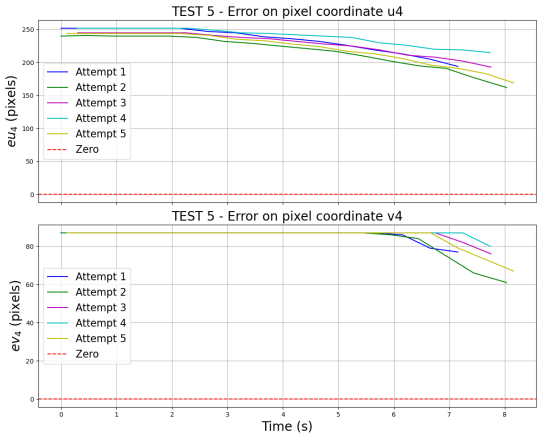


Figure 6.58: Error on feature s4 during closed loop phase for TEST5 with IS model

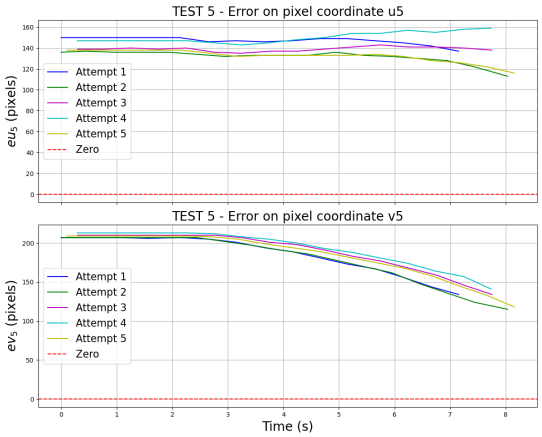


Figure 6.59: Error on feature s5 during closed loop phase for TEST5 with IS model

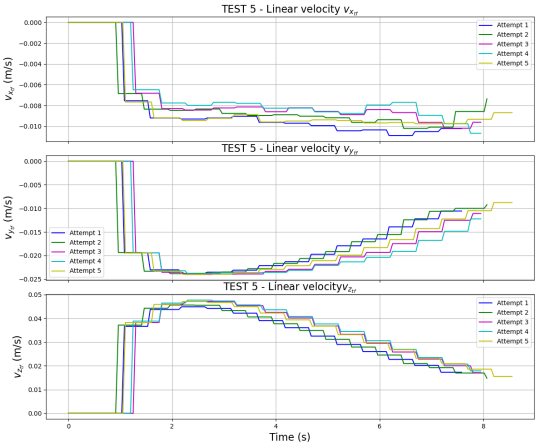


Figure 6.60: Linear velocity during closed loop phase for TEST5 with IS model

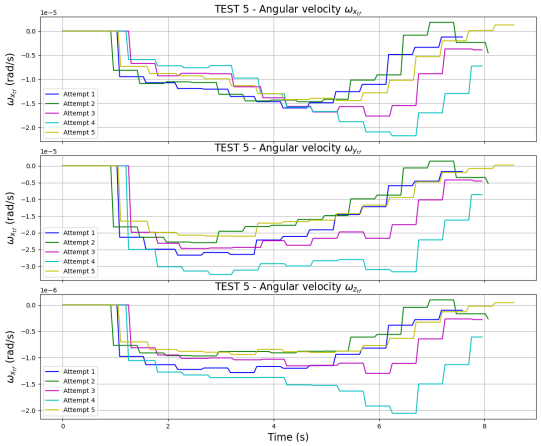


Figure 6.61: Angular velocity during closed loop phase for TEST5 with IS model

In the error plots of Test 5, it appears that the error on the pixel coordinate u of the feature s2 and on u of the feature s3 deviates from zero. However, this behavior could actually be a result of the controller correcting the error on other features.

6.2.2 Summary of Fixed initial Configuration Experiments

Tab. 6.9 summarizes the results of the first set of experiments, conducted in a controlled environment with Fixed Initial robot Configuration. The findings are

categorized into observations regarding the Vision Module, the Image-Based Visual Servo (IBVS) controller and the overall system architecture.

Vision Module Performance

The **Vision Module** demonstrated great performances during experimental tests. The Instance Segmentation (IS) model exhibited precise apple segmentation and stable feature extraction across all trials, resulting in accurate target localization. The speed and stability of the YOLOv8 architecture, combined with the OpenVino acceleration, ensured real-time processing without compromising accuracy.

A comparison between the IS model and the Multi-class Classification (MC) model reveals notable differences. The fine-tuned YOLOv8 model for IS achieved a 100% success rate (25/25 trials) in entering the Closed Loop state, as evidenced by the feature error consistently converging to zero. In contrast, Test 1, conducted with the CNN for both Multi-class Classification and Instance Segmentation, recorded a significantly lower Successful Picking Rate (SPR) of 40% (2/5), with diverging error plots. When the same test was performed using the CNN for IS only, the SPR improved to 100% (5/5), with converging error plots. This suggests that the MC model requires further refinement before being suitable for practical applications.

Image-Based Visual Servo (IBVS) Controller Performance

The IBVS controller effectively guided the robot's end effector toward the apple target when accurate object features were extracted by the Vision Module. Analysis of the error plots across all tests shows a consistent pattern: features errors began at non-zero values and converge towards zero over time. The convergence is generally faster in the initial phase, slowing as the error approaches zero. This behavior is critical to ensure smooth and precise motion of the end effector during the task.

The Euclidean error norms of the projected normalized coordinates in the image plane at steady state (end of the closed-loop phase), for each feature, are reported in Tab. 6.10. The lowest error norms, all under 0.3, were achieved in Test 2, where the apple was positioned in the center of the image plane. Interestingly, this did not directly influence the success of the subsequent open-loop grasping phase. Tests 1 and 5, which had higher error norms, still resulted in successful open-loop grasps, while Test 2 experienced two failures despite lower error norms. This suggests that the grasping phase's success is less dependent on error norms and more on other factors, such as the precision of the detected target centroid.

The velocity command data indicate that the linear velocities did not saturate the imposed limit of 0.06 m/s. The magnitude of the linear velocity components

(vx,vy,vz) varied depending on the apple's position, while the angular velocity components were much smaller (on the order of 10^{-6} to 10^{-5}) compared to linear velocities, as expected, to aid in feature alignment. The overall motion remained smooth, helped by the Exponential Moving Average (EMA) filter, though a step-like behavior was observed in the velocity plots due to the system's discrete control inputs and difference between the faster IBVS control frequency and slower inference frequency (updating the control inputs).

The average closed-loop duration (CLT) was 8.44 seconds, with a standard deviation of 1.422 seconds. This duration varied depending on the apple's distance from the tool, with closer targets resulting in faster convergence. However, the duration was also constrained by the hardware, specifically the CPU-limited inference speed. During the parameter tuning phase, increasing the gain λ (dictating the IBVS' convergence speed) led to faster convergence but introduced oscillations and reduced precision, compromising grasping success. Therefore, a moderate value of λ was chosen to balance speed and accuracy.

Overall System Architecture Performance

The overall system architecture performed as expected, with the YOLOv8-based Instance Segmentation model functioning reliably for Visual Perception. By correctly inferring the RGB data and extracting features for the IBVS controller, the system achieved a 100% success rate in completing the closed-loop phase. The IBVS controller consistently positioned the end effector toward the target apple, regardless of the apple's position in the scene.

However, the open-loop grasping phase was more sensitive to errors and depended heavily on the accuracy of the detected target centroid. A more precise centroid extraction led to more accurate open-loop velocity commands during the grasping phase. Then, also the grasping phase showed variability depending on the accuracy of the visual perception.

The overall successful picking rate (SPR) across all tests was 20/25=80%, with an average harvest time of 19.132 s and a standard deviation of 1.373 s. This extended Harvest Time is also imputable to the intended pauses between different operational phases, especially the transition from Cartesian control (used in the closed-loop and grasping phases) to joint control (used during the placing phase).

n.	CL		OL	PLACE	
	succ.	CLT	succ.	SPR	HT
1	5/5	(10.726 ± 0.23)s	5/5	5/5	(21.1 ± 0.26)s
2	5/5	(8.87 ± 0.23)s	4/5	3/5	(19.46 ± 0.47)s
3	5/5	(6.52 ± 0.36)s	5/5	3/5	(17.1 ± 0.11)s
4	5/5	(8.01 ± 0.04)s	5/5	4/5	(18.51 ± 0.14)s
5	5/5	(8.05 ± 0.35)s	5/5	5/5	(18.68 ± 0.28)s
mean	25/25	(8.44 ± 1.422)s	24/25	20/25	(19.132 ± 1.373)s

Table 6.9: Results of Fixed Initial Joint Configuration experiments

n.	$\|e(\infty)\|_{s_1}$	$\|e(\infty)\|_{s_2}$	$\|e(\infty)\|_{s_3}$	$\|e(\infty)\|_{s_4}$	$\|e(\infty)\|_{s_5}$
1 MC	0.442 ± 0.146	0.397 ± 0.171	0.434 ± 0.04	0.376 ± 0.279	0.378 ± 0.075
1 IS	0.262 ± 0.017	0.367 ± 0.019	0.28 ± 0.013	0.11 ± 0.01	0.236 ± 0.012
2	0.225 ± 0.014	0.239 ± 0.045	0.108 ± 0.03	0.073 ± 0.021	0.115 ± 0.007
3	0.325 ± 0.044	0.171 ± 0.042	0.265 ± 0.043	0.382 ± 0.049	0.292 ± 0.045
4	0.115 ± 0.009	0.268 ± 0.008	0.319 ± 0.013	0.209 ± 0.014	0.207 ± 0.007
5	0.427 ± 0.037	0.326 ± 0.028	0.175 ± 0.024	0.326 ± 0.033	0.301 ± 0.031

Table 6.10: Euclidean norm of the error for each feature at steady state for Fixed initial Configuration Experiments

6.2.3 Results of the Variable initial Configuration Experiments

To qualitatively evaluate the robustness of the system, additional experiments were conducted with variable initial robot configurations and variable apple position in the scene. The results, summarized in Table 6.11, were compared against those obtained considering only fixed initial configuration.

Tab. 6.13 confirms that the apple's position in the scene does not significantly affect the Open Loop Grasping phase, but it does influence the Euclidean error norm values on the features' projected normalized coordinates reached at the end of IBVS Closed Loop phase. Specifically, the error norm is lower when the apple is already near the center of the image plane, as less correction is required during the approaching phase. This is particularly evident in Tests 2, 3, and 4, where the apple was central in the u direction of the image plane, with only its position along the v direction varying. These tests showed lower Euclidean error norm on the normalized feature parameters. However, as in other cases, there is no clear

relationship between the minimum error reached during the Closed Loop phase and the success of the grasping phase. Even with lower error, successful grasping remains independent of the error norm achieved at the end of the Closed Loop phase.

By averaging the experimental metrics from both variable and fixed initial robot configurations as shown in Tab. 6.12, the following results were obtained:

- Successful Picking Rate (SPR): 46/60=76.66% (compared to 20/25=80% in the Fixed Initial Configuration)
- Closed Loop Time (CLT): 7.47 ± 1.69 s
- Harvest Time (HT): 18.059 ± 3.073 s

These results indicate that the designed visual servo and instance segmentation system is robust, as it performs effectively regardless of the robot's initial position and the apple's location in the scene. The slightly reduced SPR (26/35=74.28% if only considering Variable Initial Configurations experiments and 46/60=76.66% averaging Variable and Fixed Initial configuration experiments, compared to 20/25=80% in the Fixed configuration tests) remains high and falls within the range of state-of-the-art systems, which typically achieve SPR values between 60% and 80%. This suggests that the system maintains a high degree of reliability even under variable initial conditions.

In terms of Closed Loop performance, the average CLT of (6.78 ± 1.53) s considering Variable Configuration experiments and (7.47 ± 1.69) s averaging Fixed and Variable Configuration experiments, is slightly reduced with respect to that of the Fixed configuration experiments. This strictly depends on the setup and suggests that the controller's performance remains robust even when starting from variable robot positions, confirming the system's adaptability across varying initial conditions.

The average Harvest Time for Variable Configuration experiments (17.23 ± 3.65) s and overall variable and fixed configuration experiments (18.059 ± 3.073) s were slightly reduced, likely due to variations in the end effector's initial distance from the apple. Although the overall HT remains higher than the 6 to 8 seconds typically reported by state-of-the-art systems, the primary constraint is the CPU-limited inference speed in the Vision Module, which can create a bottleneck during the Closed Loop phase. Upgrading to a more powerful CPU is expected to significantly reduce this phase's duration by speeding up the real-time inference and improving overall system responsiveness.
In addition, the Open Loop Grasping phase takes time, being currently tuned to 2.4 s to avoid abrupt accelerations and unwanted shifts, which could otherwise displace the mockup apple and lead to grasping failures. The integration of a motion planner

by MoveIt 2 would improve the precision of the grasping, particularly by ensuring smoother movements. Lastly, optimizing the transition from Cartesian velocity commands to joint velocity commands would reduce the time taken for the placing phase, thereby further lowering the total HT and improving the system's efficiency in pick-and-place operations.

n.	JOINT CONF.	CL		OL	PLACE	
		succ.	CLT	succ.	SPR	HT
1	-63°, -18°, 96°, 3°, -100°, -22°	5/5	(5.23 ± 0.832)s	4/5	4/5	(16.525 ± 0.36)s
2	-34°, -15°, 86°, 4°, -85°, 13°	5/5	(6.84 ± 0.641)s	4/5	3/5	(19.66 ± 0.256)s
3	-58°, -34°, 30°, 18°, -91°, -19°	5/5	(4.06 ± 0.413)s	3/5	0/5	NV
4	-59°, -28°, 90°, 5°, -63°, -30°	5/5	(7.62 ± 0.51)s	5/5	5/5	(18.41 ± 0.52)s
5	-44°, -27°, 63°, 22°, -90°, -14°	5/5	(6.03 ± 0.41)s	5/5	5/5	(16.458 ± 0.39)s
6	-30°, -6°, 85°, 1°, -93°, -7°	5/5	(8.31 ± 0.249)s	5/5	5/5	(18.87 ± 0.145)s
7	-71°, -9°, 91°, -7°, -92°, -9°	5/5	(8.724 ± 0.482)s	5/5	4/5	(19.745 ± 0.514)s
mean		35/35	(6.78 ± 1.53)s	31/35	26/35	(17.23 ± 3.65)s

Table 6.11: Results of Variable Initial Joint Configuration experiments

Tests	CL		OL	PLACE	
	success	CLT	success	SPR	HT
mean of Fix. and Var. Ini. conf.	60/60	(7.47 ± 1.69)s	55/60	46/60	(18.059 ± 3.073)s

Table 6.12: Results of overall experiments

n.	$\|e(\infty)\|_{s_1}$	$\|e(\infty)\|_{s_2}$	$\|e(\infty)\|_{s_3}$	$\|e(\infty)\|_{s_4}$	$\|e(\infty)\|_{s_5}$
1	0.529 ± 0.067	0.275 ± 0.004	0.328 ± 0.011	0.559 ± 0.068	0.401 ± 0.031
2	0.219 ± 0.016	0.181 ± 0.013	0.072 ± 0.015	0.193 ± 0.015	0.075 ± 0.003
3	0.325 ± 0.034	0.194 ± 0.017	0.088 ± 0.008	0.279 ± 0.03	0.146 ± 0.021
4	0.038 ± 0.019	0.1571 ± 0.03	0.265 ± 0.03	0.279 ± 0.014	0.121 ± 0.024
5	0.522 ± 0.073	0.405 ± 0.063	0.223 ± 0.028	0.399 ± 0.036	0.372 ± 0.039
6	0.457 ± 0.087	0.611 ± 0.141	0.494 ± 0.1	0.296 ± 0.007	0.443 ± 0.073
7	0.192 ± 0.018	0.341 ± 0.009	0.325 ± 0.009	0.163 ± 0.016	0.234 ± 0.013

Table 6.13: Euclidean norm of the error for each feature at steady state for Variable Initial Configuration Experiments

Chapter 7

Conclusion and Future Developments

The thesis work aimed at designing a Visual Servo for apple harvesting using the Kinova Gen 3 Lite robot, equipped with an Intel RealSense D435i camera mounted on its end effector. The proposed architecture included a Vision Module, integrating a YOLOv8 fine-tuned Instance Segmentation network, responsible for apple identification and visual features extraction from the selected target object, and an Image-Based Visual Servo Block, tasked with guiding the Kinova's end-effector toward the target apple by directly computing the error between current and desired features in the image space.
The proposed architecture was implemented as a Finite State Machine, to evaluate the performance of individual modules and their combined efficacy. Extensive testing of the system in realistic scenario, recreated at PIC4SeR laboratory, revealed the following findings. Firstly, the fine-tuned YOLOv8 CNN for Instance Segmentation outperformed the fine-tuned model devoted to both Instance Segmentation and Multi-class Classification of apples based on their degree of occlusion. It enabled accurate segmentation of mockup apples and precise visual feature extraction for the visual controller. Further refinement of the YOLOv8 CNN for both Instance Segmentation and Multi-class Classification tasks will be necessary to utilize the network in practical applications.
Secondly, the IBVS consistently and smoothly navigated the end-effector toward the target when visual features extraction was accurate and without any loss or misidentification by the YOLOv8 network. The final open-loop control phase proved crucial in overcoming the IBVS's inability to compute the control law when the target's depth became undetectable by the RGBD sensor, ultimately leading to successful grasping.
Lastly, the system fulfilled the complete autonomous harvesting task, achieving the

competitive Successful Picking Rate (SPR) of 80% in Fixed Initial Configuration experiments, 74.28% in Variable Initial Configuration experiments and 76.66% when averaging results from Fixed initial and Variable initial Configuration experiments, reaching an average Harvest Time of respectively 19.132 ± 1.373 s, 17.23 ± 3.65 s and 18.059 ± 3.073s. Beyond the achieved goals, the extended harvest time and collisions as the primary source of failure during testing highlight potential areas for future developments:

- attempting to considerably accelerate the inference time, e.g. by employing dedicated GPU, to achieve near real-time control, as the current 5 fps inference speed is significantly lower than the 30 fps frame rate of the RGB sensor. This would speed up the convergence of visual control;
- integrating the MoveIt 2 motion planner to execute grasping as soon as depth is lost by the RGBD sensor and the IBVS control law is no longer computable. This would significantly enhance the speed and precision of the pick and place operation;
- incorporating a collision checking and obstacle avoidance model to improve system efficiency in real-world scenarios.

Appendix A

Adam optimizer

The Adam (Adaptive Moment Estimation) optimizer is a popular algorithm used for training deep learning models. It combines the advantages of two other methods: AdaGrad, which maintains per-parameter learning rates for dealing with sparse gradients, and RMSProp, which adjusts learning rates based on the average of recent magnitudes of gradients. Adam computes individual adaptive learning rates for each parameter by maintaining two moment estimates: the first moment (mean) m_t and the second moment (uncentered variance) v_t.
The updates for parameters θ at time step t are computed as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where g_t is the gradient at time step t, β_1 and β_2 are the decay rates for the moment estimates (typically set to 0.9 and 0.999, respectively). To correct for initialization bias, Adam applies bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, the parameter update rule is given by:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where η is the learning rate and ϵ is a small constant for numerical stability. Adam is computationally efficient, has little memory requirement and works well for problems with noisy or sparse gradients [44].

Appendix B

AdamW optimizer

AdamW is a variant of the Adam optimizer that decouples weight decay from the gradient update, addressing a key issue in Adam, where L2 regularization (often implemented as weight decay) is implicitly applied to both the gradient and the adaptive learning rates. In AdamW, weight decay is applied directly to the weights during the update step, without affecting the gradient-based parameter update, which improves generalization in deep learning models. In AdamW, the parameter update is modified as:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right)$$

where λ represents the weight decay coefficient. This decoupling leads AdamW to better convergence and improved performance, especially in models with very deep architectures. The introduction of AdamW has been particularly impactful in large-scale models like Transformers, where AdamW is now a standard choice for optimization [45].

Appendix C

Grid Search

Grid Search is a hyperparameter optimization technique widely used in machine learning to enhance model performance. The process involves defining a grid of possible hyperparameter values and then evaluating the model for each combination of these parameters. By systematically testing every combination, Grid Search ensures that the best hyperparameters are selected based on a predefined evaluation metric such as accuracy, F1 score, mean Average Precision, or mean squared error. However, Grid Search can be computationally expensive, particularly when the hyperparameter space is large, as the number of combinations grows exponentially with the number of hyperparameters. This method is particularly effective when the search space is limited and well-understood, as it guarantees that all possible combinations are tested, leaving no potential configurations unexplored.

Appendix D

Metrics for Object Detection and Classification

- As first main metric to evaluate an Object Detection and Instance Segmentation model, we find the **Intersection over Union (IOU)** [22]:

$$IOU = \frac{area\left(\hat{B} \cap B\right)}{area\left(\hat{B} \cup B\right)}$$

where $\hat{B}$ is the predicted bounding box or mask, B is the ground-truth bounding box or mask, $\cap$ is the intersection symbol and $\cup$ is the union symbol. Fig. D.1 represents the graphical meaning of IOU.

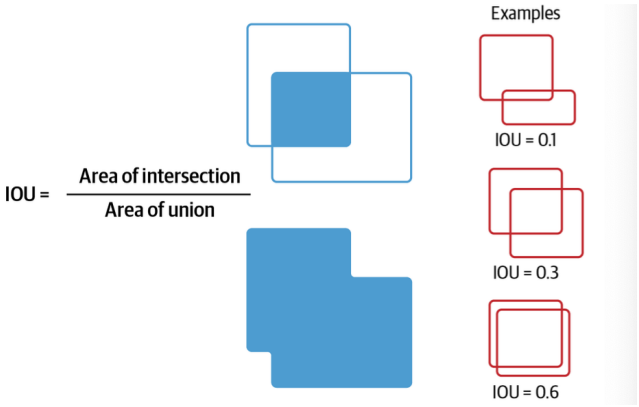


Figure D.1: Representation of Intersection over Union from [22]

- **Precision** metric aims at measuring how well the model identifies only the

correct objects in the image and it is computed as [22]:

$$precision = \frac{TP}{TP + FP} = \frac{TP}{alldetections}$$

- **Recall** metric estimates the model's ability to correctly detect all the positive instances from all the actual positive ones [22]:

$$recall = \frac{TP}{TP + FN} = \frac{TP}{allactualboundingboxes}$$

- The mean Average Precision (**mAP**) indicates if the model is able to correctly classifying objects across multiple classes: it measures how well the model balances precision and recall across different classes. In the discrete form, mAP is computed by averaging the Average Precision (AP) across all object categories.

  The Average Precision (AP) for a single class is calculated by summing over the differences in recall values multiplied by the corresponding interpolated precision:

$$AP = \sum_{n=1}^{N} (r_n - r_{n-1})\, p_{\text{interp}}(r_n)$$

  where r_n is the recall at the n-th threshold, $p_{\text{interp}}(r_n)$ is the interpolated precision at that recall level, N is the total number of thresholds used for precision and recall computation.

  To compute the mAP, we average the AP values over all C object classes:

$$mAP = \frac{1}{C} \sum_{i=1}^{C} AP_i$$

  where C is the total number of object classes, AP_i is the Average Precision for the i-th class.

Bibliography

[1] Catherine Bernier. *Harvesting Robots: Automated Farming in 2023*. Mar. 2023. URL: https://howtorobot.com/expert-insight/harvesting-robots (cit. on pp. 3, 4).

[2] Hongyu Zhou, Xing Wang, Wesley Au, Hanwen Kang, and Chao Chen. «Intelligent robots for fruit harvesting: Recent developments and future challenges». In: *Precision Agriculture* 23.5 (2022), pp. 1856–1907 (cit. on pp. 5, 7, 8).

[3] Lei Li, Qin Zhang, and Danfeng Huang. «A review of imaging techniques for plant phenotyping». In: *Sensors* 14.11 (2014), pp. 20078–20111 (cit. on p. 6).

[4] Hanwen Kang, Hongyu Zhou, and Chao Chen. «Visual perception and modeling for autonomous apple harvesting». In: *IEEE Access* 8 (2020), pp. 62151–62163 (cit. on p. 7).

[5] Hanwen Kang, Hongyu Zhou, Xing Wang, and Chao Chen. «Real-time fruit recognition and grasping estimation for robotic apple harvesting». In: *Sensors* 20.19 (2020), p. 5670 (cit. on pp. 7, 8).

[6] Harvest Croo Robotics. *Harvest Croo Robotics*. Accessed: 2024-09-03. 2024. URL: https://www.harvestcroorobotics.com (cit. on p. 8).

[7] C Wouter Bac, Eldert J Van Henten, Jochen Hemming, and Yael Edan. «Harvesting robots for high-value crops: State-of-the-art review and challenges ahead». In: *Journal of field robotics* 31.6 (2014), pp. 888–911 (cit. on p. 8).

[8] Kaixiang Zhang, Kyle Lammers, Pengyu Chu, Zhaojian Li, and Renfu Lu. «An automated apple harvesting robot—From system design to field evaluation». In: *Journal of Field Robotics* (2023) (cit. on pp. 9, 10).

[9] Tao Li, Feng Xie, Zhuoqun Zhao, Hui Zhao, Xin Guo, and Qingchun Feng. «A multi-arm robot system for efficient apple harvesting: Perception, task plan and control». In: *Computers and Electronics in Agriculture* 211 (2023), p. 107979 (cit. on pp. 9, 10).

[10] Agromillora. *A dream come true: The fresh fruit picking robot*. Aug. 2022. URL: https://www.agromillora.com/olint/en/a-dream-come-true-the-fresh-fruit-picking-robot/ (cit. on p. 11).

[11] FFRobotics. *FFRobotics*. 2020. URL: https://www.ffrobotics.com (cit. on p. 11).

[12] Tevel. *Tevel Technology*. 2017. URL: https://www.tevel-tech.com/home/ (cit. on p. 12).

[13] Francois Chaumette and Seth Hutchinson. «Visual servo control. I. Basic approaches». In: *IEEE Robotics Automation Magazine* 13.4 (2006), pp. 82–90. DOI: 10.1109/MRA.2006.250573 (cit. on pp. 13, 19, 20, 24, 25, 47).

[14] Mohammad Keshmiri. «Image based visual servoing using trajectory planning and augmented visual servoing controller». PhD thesis. Concordia University, 2014 (cit. on pp. 13–17).

[15] Akshay Kumar. *An Overview of Visual Servoing for Robot Manipulators*. https://control.com/technical-articles/an-overview-of-visual-servoing-for-robot-manipulators/. 2020 (cit. on p. 15).

[16] Danica Kragic, Henrik I Christensen, et al. «Survey on visual servoing for manipulation». In: *Computational Vision and Active Perception Laboratory, Fiskartorpsv* 15 (2002), p. 2002 (cit. on pp. 15, 16).

[17] Alessandro De Luca. *Visual Servoing*. Lecture Notes. Robotics 2, DIAG Robotics Laboratory, Sapienza University of Rome (cit. on p. 16).

[18] Peter I Corke, Witold Jachimczyk, and Remo Pillat. *Robotics, vision and control: fundamental algorithms in MATLAB*. Vol. 73. Springer, 2011 (cit. on p. 21).

[19] Seth Hutchinson, Nicholas Gans, Peter Corke, and Sourabh Battacharya. *Visual Servo Control*. PowerPoint Presentation. Includes material from articles in IEEE Robotics and Automation Society Magazine: "Visual Servo Control, Part I: Basic Approaches" and "Visual Servo Control, Part II: Advanced Approaches," by François Chaumette and Seth Hutchinson (cit. on pp. 23, 24).

[20] Ezio Malis. «Improving vision-based control using efficient second-order minimization techniques». In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*. Vol. 2. IEEE. 2004, pp. 1843–1848 (cit. on p. 25).

[21] Patrick Langechuan Liu. «Single Stage Instance Segmentation — A Review». In: *Towards Data Science* (2020). URL: https://towardsdatascience.com/single-stage-instance-segmentation-a-review-1eeb66e0cc49 (cit. on p. 26).

[22] Valliappa Lakshmanan, Martin Görner, and Ryan Gillard. *Practical machine learning for computer vision*. " O'Reilly Media, Inc.", 2021 (cit. on pp. 26, 37, 113, 114).

[23] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015 (cit. on pp. 26, 31–36).

[24] Jonas Teuwen and Nikita Moriakov. «Convolutional neural networks». In: *Handbook of medical image computing and computer assisted intervention*. Elsevier, 2020, pp. 481–501 (cit. on pp. 27–29, 32, 34–36).

[25] VoiceVibes. *What is a Convolutional Neural Network, CNNs?* https://medium.com/@VoiceVibes_101/what-is-a-convolutional-neural-network-cnns-56139c1f1aaf. 2023 (cit. on p. 28).

[26] A Géron. «Hands-On machine learning with scikit-learn, keras & tensorflow farnham». In: *Canada: O'Reilly* (2019) (cit. on pp. 28, 29, 36).

[27] Zoumana Keita. *An introduction to Convolutional Neural Networks (CNNs)*. https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns. 2023 (cit. on p. 32).

[28] Jane Torres. *YOLOv8 Architecture Explained: Exploring the YOLOv8 Architecture*. https://yolov8.org/yolov8-architecture-explained/. 2024 (cit. on p. 37).

[29] Ultralytics. *Ultralytics*. URL: https://docs.ultralytics.com (cit. on pp. 37, 68).

[30] Jacob Solawetz and Francesco. *What is YOLOv8? The Ultimate Guide*. https://blog.roboflow.com/whats-new-in-yolov8/. 2023 (cit. on p. 38).

[31] Juan Pedro. *Detailed Explanation of YOLOv8 Architecture (Part 1)*. https://medium.com/@juanpedro.bc22/detailed-explanation-of-yolov8-architecture-part-1-6da9296b954e. 2023 (cit. on p. 39).

[32] Kinova Robotics. *GEN3 Lite Robots*. URL: https://www.kinovarobotics.com/product/gen3-lite-robots (cit. on pp. 56, 57).

[33] Kinova Robotics. *Gen3 Lite User Guide*. Accessed: 2024-09-18. 2024. URL: https://artifactory.kinovaapps.com/ui/api/v1/download?repoKey=generic-public&path=Documentation%2FGen3%20lite%2FTechnical%20documentation%2FUser%20Guide%2FGen3_lite_USER_GUIDE_R03.pdf (cit. on pp. 57, 59, 60).

[34] Kinova Robotics. *ROS 2 Kortex*. URL: https://github.com/Kinovarobotics/ros2_kortex (cit. on p. 60).

[35] Intel. *Intel RealSense Depth Camera D435i*. URL: https://www.intelrealsense.com/depth-camera-d435i/ (cit. on pp. 60, 61).

[36] Francisco Martín Rico. *A concise introduction to robot programming with ROS2*. Chapman and Hall/CRC, 2022 (cit. on pp. 61–63).

[37] Open Robotics. *ROS 2 Humble Tutorials*. 2024. URL: https://docs.ros.org/en/humble/Tutorials/ (cit. on pp. 63, 64).

[38] Open Robotics. *ROS 2 Humble Concepts*. 2024. URL: https://docs.ros.org/en/humble/Concepts.html (cit. on pp. 65, 66).

[39] Open Robotics. *Gazebo*. URL: https://gazebosim.org/home (cit. on p. 66).

[40] PickNik Robotics. *MoveIt 2 Documentation*. URL: https://moveit.picknik.ai/humble/index.html (cit. on pp. 66, 67).

[41] PickNik Robotics. *Realtime Arm Servoing*. URL: https://moveit.picknik.ai/humble/doc/examples/realtime_servo/realtime_servo_tutorial.html (cit. on p. 67).

[42] Ultralytics. *Ultralytics OpenVINO Integration Documentation*. URL: https://docs.ultralytics.com/it/integrations/openvino/ (cit. on p. 68).

[43] Roboflow. *Roboflow: Create and Deploy Computer Vision Models* (cit. on p. 69).

[44] Diederik P Kingma. «Adam: A method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 110).

[45] Ilya Loshchilov and Frank Hutter. «Decoupled Weight Decay Regularization». In: *International Conference on Learning Representations*. 2017. URL: https://api.semanticscholar.org/CorpusID:53592270 (cit. on p. 111).