SAPIENZA
UNIVERSITÀ DI ROMA

Design and Implementation of a Novel Grasping System for a Humanoid Robot in QiBullet Simulator

Faculty of Information Engineering, Informatics and Statistics
Master Degree Program in Computer Science

Candidate
Jessica Frabotta
ID number 1758527

Thesis advisor
Dott. Luigi Cinque

Academic Year 2024/2025

Design and Implementation of a Novel GraspingSystem for a Humanoid Robot in QiBullet Simulator

Master's thesis.. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: frabotta.1758527@studenti.uniroma1.it

Dedicato a
Ciambi e alla nostra unica fede e unica passione

Abstract

Grasping 3D objects reliably remains one of the most complex challenges in robotics. It involves balancing multiple factors, including hand kinematics, object geometry, material properties, applied forces and environmental constraints. This complexity creates an enormous space of potential grasps, making exhaustive real-world testing impractical. With so many variables, there are countless ways a robot could try to grasp an object, and testing every single one in real life is just not practical. That's where simulation comes in. By planning and testing grasps in a virtual environment, we can explore and optimize strategies without the risks and costs of real-world experiments.

This thesis focuses on the design and implementation of a grasping system for unknown objects using a humanoid robot within the QiBullet simulator. Since the objects are unknown, the robot must first detect and localize them within the environment. To achieve this, the system integrates state-of-the-art object detection and monocular depth estimation techniques—specifically, YOLO (You Only Look Once) for object detection and Depth Anything for depth estimation—along with established robotics methods, such as inverse kinematics, to enable single-handed grasp execution. Although grasping is not typically a critical skill for humanoid robots, which are often deployed in social or indoor settings, enhancing this capability represents a significant technical challenge and expands their potential applications.

The proposed approach has demonstrated robustness and reliability, making it a strong candidate for real-world testing on a physical robot. The Aldebaran Pepper robot was selected as the platform for this work.

This is the first method we have identified in the literature that enables Pepper to complete the grasping task without using any markers or additional sensors beyond those already provided by default with the robot. The system is implemented in Python, primarily using Aldebaran's Python SDK, with the Robot Operating System (ROS) handling the inverse kinematics implementation.

Extensive experiments were conducted to validate the effectiveness and robustness of the approach, which can be easily adapted to other similar robots.

Contents

Chapter 1

Introduction

This chapter is organized as follows. Section 1.1 presents the challenges and problems that characterize the task addressed in the thesis. Section 1.2 outlines the objectives of the thesis. Finally, in Section 1.3, the contribution of the proposed work is highlighted and the structure of the thesis is also provided.

1.1 Problem statement

The human body is regarded as one of the most sophisticated structures on earth, with the human hand being a highly exquisite manipulator that can perform everything from the delicate touch of fragile objects to the power grasping of a bench press. Unlike other primates, the human hand can oppose the thumb to the index finger and the other fingers: the fine movement of the fingers allows the use of very small and thin tools. In the human hand, different grip modes can be adapted to a wide range of tasks, exploiting different configurations of fingertips, joints and phalanges (Fig. 1.1).

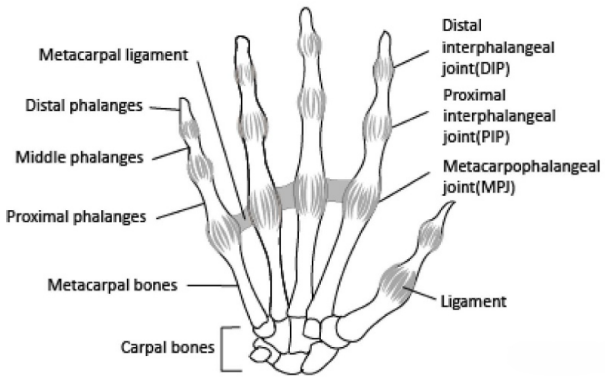

Figure 1.1 Illustration of human hand bone structure [1].

In general, as shown in Fig. 1.2, human grasping methods can be divided into two categories: *pinching grasp* and *enveloping grasp*. The first category is typically used for thin or small objects, which require precision and dexterity operations. The

second, however, is suitable for bulky or heavy objects, which require a stable grip capable of resisting external forces in different directions.

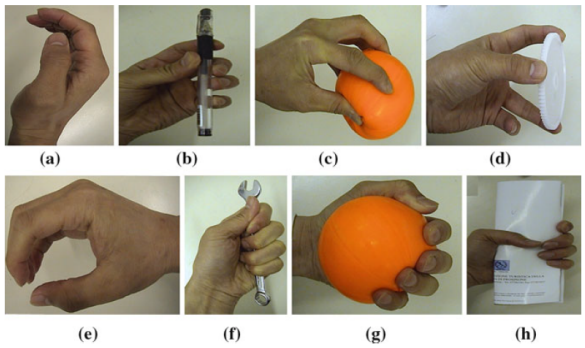

Figure 1.2 A grasp taxonomy of human hand: **(a)** pinching grasp, **(b)** grasp a pen, **(c)** grasp a sphere, **(d)** grasp circle, **(e)** enveloping grasp, **(f)** grasp a spanner, **(e)** grasp a sphere and **(h)** grasp a book [2].

In real-world applications, humans can determine the number of contact points needed and the most suitable grip configuration based on the shape, size, and nature of the object they are handling. This adaptability comes from the complex and flexible structure of our fingers and palms. Through experiments on human grasping, researchers have mapped out how the geometric features of objects—like width, length, height, and shape—relate to the most effective ways to grip them.

For example, as shown in Fig. 1.3, objects with circular, spherical, cylindrical, or oblong shapes require different approaches. These can range from a *parallel grasp*, using two or three fingers aligned side by side, to a *centripetal grasp*, where all fingers wrap around the object's surface.

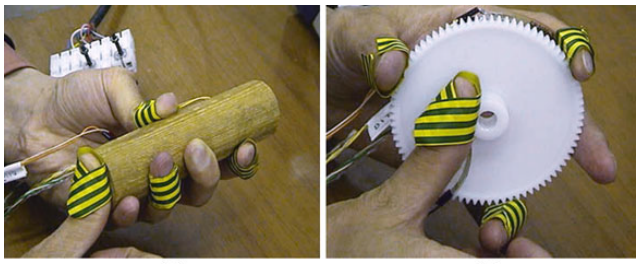

Figure 1.3 Experiments with human grasping: **(a)** three-finger parallel pinching grasp of a cylinder made of wood and **(b)** three-finger centripetal pinching grasp of plastic gear [2].

The attention of the robotics community has been drawn more and more to humanoid robots in the last years. Humanoid robots are designed to mimic the appearance and behavior of humans and to perform specific tasks in conjunction with or instead of humans (Fig. 1.4). Today's humanoid robots come in different shapes and sizes and are extensively being used for research and space exploration, personal assistance and care-giving, education and entertainment, search and rescue operations, manufacturing and maintenance, public relations, and most importantly

healthcare sector. Their design, building and applications addresses many interesting research challenges: biped walking, human-robot interaction, autonomy, interaction with unstructured and unknown environments, and many others. Among them, the development of manipulation skills is of utmost importance and one of the most complex.

Figure 1.4 The humanoid robot ARMAR-IIIa working in a kitchen environment [3].

One of the main challenges that humanoid developers have to face when considering manipulation issues is the design of robot hands and arms. In the case of hands for humanoids, the design is guided by the need for great versatility, which means a large number of fingers and degrees of freedom, reduced size and human-like appearance. A constant issue has been to design human-size light arm/hand systems either focusing on a pure mechanical approach [4] or taking some anthropomorphic and biological inspiration [1]. The force transmission method is one of the main factors affecting the weight and output of the robotic upper limb (Fig. 1.5).

(a) (b) (c) (d)

Figure 1.5 Some examples of robotic hands with different force transmission methods: **(a)** linkage-driven [5], **(b)** tendon-driven [6], **(c)** pneumatic [7], **(d)** SMA [8].

The *linkage-driven* system relies on rigid mechanical connections and was once popular for its straightforward design and movement precision. However, these systems lack compliance, which is the ability to flex elastically under external forces and return to their original shape once the force is gone. *Tendon-driven systems* are currently the most common architecture for robotic hands. They use cables (tendons) to control fingers, offering precise movement with added flexibility, making them

ideal for handling objects of various shapes. They offer similar positional control to linkage-driven system but with compliance. *Pneumatic systems* use compressed air for movement, allowing soft, adaptable grips. The downside of this approach is that sophisticated controls are needed to manage air pressure and flow. Lastly, probably the most innovative approaches are *SMA (Shape Memory Alloy)* and *SMP (Shape Memory Polymer)* which use materials that change shape with heat or electricity. The challenges related to these methods are response time (latency) and longevity, as they can degrade with repeated use.

Creating robotic hands and arms that are increasingly similar to human ones, both in appearance and functionality, is not the only reason why the task of grasping in robotics is so complex. If we consider manipulation in humanoid robotics, we can describe the general process with the scheme in Fig. 1.6. As shown, a large amount of information and strategies is needed to implement the grasping system. This scheme is divided into three macro-areas:

- *anthropomorphic perception*, which collects stimuli from the environment through sensors inspired by human ones and carries out a first level of processing;
- *anthropomorphic processing*, which interprets sensory information using techniques that reproduce human reasoning and behavioral planning;
- *anthropomorphic action*, which allows interaction with the environment thanks to actuators controlled to imitate human movements.

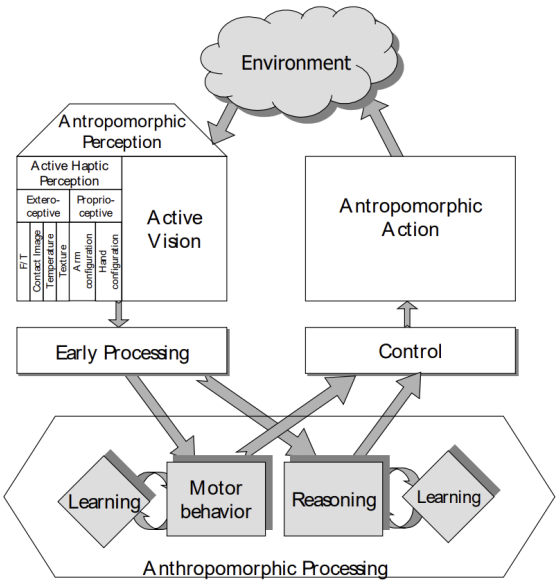

Figure 1.6 General manipulation process in humanoid robotics [9].

1.2 Aims and scope

In this thesis, the aim is to develop a grasping system for a humanoid robot. The system is designed for indoor environments, typical settings in which humanoid robots are used. In addition to addressing robotics challenges related to optimizing motion planning and the grasping of a target object, various aspects of computer vision have also been tackled. The main goal has been to realize a system able to grasp unknown objects without the use of markers or sensors other than cameras. Another key aspect was the design of the system on a simulator, in order to optimize its performance before testing it on a physical humanoid robot.

This approach offers several advantages. First, it allows testing many possible scenarios without risking damage to the physical robot or the surrounding environment. It also provides precise control over parameters such as object positioning, hand kinematics, and external forces, which can be difficult to replicate consistently in real-world settings. Simulators also provide the ability to visualize the various reference systems chosen, facilitating the analysis and optimization of the grasping process from different perspectives. Clearly, all this allows saving considerable time and resources during the development phase, favoring the design of a more performing system.

The humanoid robot chosen as the platform for the system is the popular Pepper robot from Aldebaran Robotics (now SoftBank Robotics). This choice is motivated by two main reasons: first, the VisionLab [10] has a physical version of this robot, which allows to verify the efficiency of the system in a real environment; second, the aim was to contribute to the state of the art in this field, as there are not many studies in the literature on robots with similar characteristics to Pepper. Unlike many other humanoid robots, Pepper has not been developed to carry out heavy duty tasks or grasping objects precisely. Instead of that, Pepper is intended to be deployed at indoor environments and has a clear social appeal.

Advances in the field of grasping can open new frontiers for humanoid robots, enabling the development of numerous new applications even for older models like Pepper. In settings such as offices, elderly care facilities, schools, or hospitals, a humanoid robot can thus not only be utilized for its interaction capabilities but also be employed in other tasks that require grasping and moving objects of varying weight and shape, serving as an assistant to humans.

Specifically, this work aims to achieve the following objectives:

- Study of the most advanced techniques and approaches to solve the problem of grasping in humanoid robots.
- Development of a reaching and grasping strategy for a humanoid robot.
- Implementation of the system on a simulator, recreating a realistic indoor environment that takes into account possible physical variations, such as lighting, object weight and the dimensions of the objects present in the simulation.

1.3 Contribution and thesis outline

By simply using a robot's camera, it is possible to obtain a large amount of information useful for recognizing and determining the position of objects in an environment. This data allows us to understand the necessary actions to move a humanoid robot closer to a target object and configure its joints to perform a single-handed grasp. Starting from this idea, the aim of this thesis is to present a new approach to realize a grasping system independent of the target object and the environment in which the robot operates (provided it is an indoor environment). State-of-the-art strategies based on deep learning were combined with classical robotics techniques. In particular, the use of the innovative and recent depth estimation model Depth Anything v2 enabled the development of an alternative approach to traditional grasp detection algorithms. The structure of the thesis is organized into the following chapters:

- **Chapter 2**: the theoretical foundation of the work is described, and the state of the art regarding the various steps that compose the grasping task on humanoid robots is presented;
- **Chapter 3**: the platforms used for the development of the system are presented, including the Pepper robot, the QiBullet simulator, and the other frameworks needed for the implementation;
- **Chapter 4**: the architecture of the proposed system is described, with a focus on the neural networks used and the algorithms developed to manage the robot kinematics;
- **Chapter 5**: the tests and experiments conducted to validate the system are illustrated;
- **Chapter 6**: conclusions are drawn, and potential directions for future developments are discussed.

Chapter 2

State of the art

The literature that has been studied during the preparation phase of this project is presented in this chapter. In particular, in Section 2.1, a general schema of the parts that usually compose a complete robotic grasping system is presented. Section 2.2 describes two fundamental concepts in robotics: forward kinematics and inverse kinematics. In Sections 2.3, 2.4, 2.5, the three subsystems presented in Section 2.1 are described in detail. Finally, in Section 2.6, similar works on some humanoid robots are presented.

2.1 General framework of a robotic grasping system

Generally speaking, a complete robotic grasping system mainly includes three parts [11] as shown in Fig. 2.1

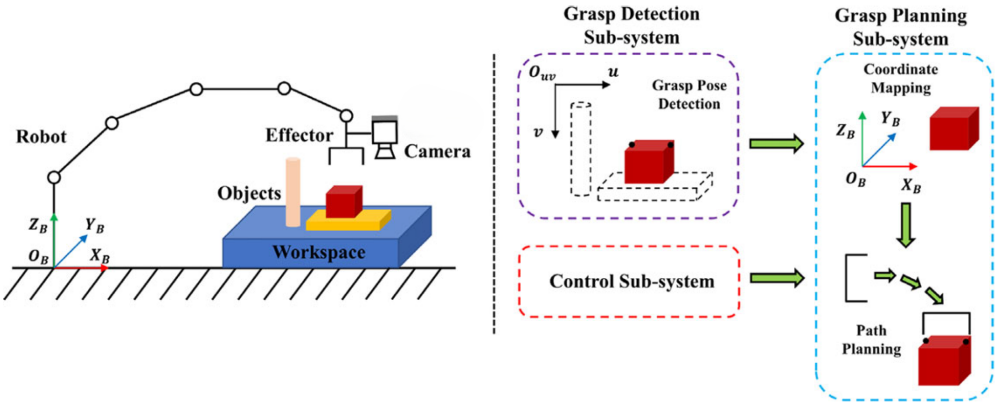

Figure 2.1 The robotic grasping system. Left: the robot is equipped with a camera and end-effector for grasping target objects in the workspace. Right: the whole system mainly includes three parts: the grasp detection subsystem, the grasp planning subsystem, and the control subsystem [12].

- **Grasp detection subsystem**: grasp detection is a visual recognition problem in which the robot uses its sensors to detect graspable objects in its environment. Advanced systems typically employ 3D vision systems and RGB-D cameras for

environmental perception. However, with recent advancements in computer vision and deep learning, simpler RGB cameras can now also be utilized. The main goal is to extract the object's position and orientation within the image coordinate system.

- **Grasp planning subsystem**: once the object has been identified and its position determined in the image, it is necessary to convert this information into the robot's coordinate system. This process, known as coordinate mapping, allows the robot to understand the object's location within its workspace. Subsequently, the system generates a feasible path for the robotic arm to reach the object.
- **Control subsystem**: in this phase, inverse kinematics equations are solved to determine the necessary movements for the robotic arm to execute the grasp correctly and control the robot to execute the grasp according to the solution results.

Before exploring each subsystem, it is important to discuss two key concepts in robotics that have been mentioned in this section: inverse kinematics and forward kinematics.

2.2 Forward kinematics and inverse kinematics

Kinematics is the science of motion that treats motion without regard to the forces which cause it. Hence, the study of the kinematics of manipulators refers to all the geometrical and time-based properties of the motion [13].

A manipulator consists of a series of rigid bodies (links) connected by joints. The whole structure forms a *kinematic chain*. One end of the chain is constrained to a base. An *end-effector* (gripper, robotic hand) is connected to the other end allowing manipulation of objects in space. *Serial chain manipulators* are a type of kinematic chain in which the links and joints are arranged in sequence.

The simplest types of joints are *revolute joints* and *prismatic joints*. Conventional representations of the two types of joints are sketched in Fig. 2.2.

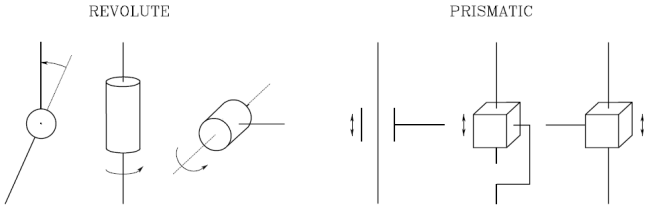

Figure 2.2 Conventional representations of joints [14].

A revolute joints works like a hinge and allows relative rotation about a single axis, while a prismatic joint allows linear movement along a single axis, that is, extension or retraction. These joints have only one degree of freedom for movement: the rotation angle in the case of a revolute joint and the amount of linear displacement

in the case of a prismatic joint. Assuming that each joint has only one degree of freedom, the action of each joint can be described by a single real number: the rotation angle in the case of a revolute joint or the displacement in the case of a prismatic joint.

The objective of *forward kinematic* analysis is to determine the cumulative effect of the entire set of joint variables, that is, to determine the position and orientation of the end-effector given the values of these joint variables. The objective of *inverse kinematic* analysis is, in contrast, to determine the values for these joint variables given the position and orientation of the end-effector frame [15].

Consider a serial chain manipulator with m joints, and let $\mathcal{W} \subseteq \mathbb{R}^n$ denote its *workspace* in a space of dimension n. The workspace represents that portion of the environment the manipulator's end-effector can access. Its shape and volume depend on the manipulator structure as well as on the presence of mechanical joint limits. We can formally define:

- Forward kinematics as the function that maps a joint configuration from the set $\mathcal{Q} \subseteq \mathbb{R}^m$ of feasible joint positions to a Cartesian position in the workspace $\mathcal{W}$:

$$f : \mathcal{Q} \to \mathcal{W} \tag{2.1}$$
$$\mathbf{q} \mapsto \mathbf{x} \tag{2.2}$$

- Inverse kinematics as the inverse mapping of the forward kinematics, i.e., the function that maps a position in the workspace to a joint configuration:

$$h : \mathcal{W} \to \mathcal{Q} \tag{2.3}$$
$$\mathbf{x} \mapsto \mathbf{q} \tag{2.4}$$

Note that h may have multiple solutions for a given $\mathbf{x}$. Finding a closed-form expression for h can be highly complex or even impossible. The inverse kinematics problem is clearly not as simple as the forward kinematics one. The forward kinematic problem has a unique solution, and its success depends on whether the joints are allowed to do the desired transformation. When dealing with inverse kinematics, it is not always the case that a solution can be achieved. There are instances where the goal is unreachable or when two or more tasks conflict and cannot be satisfied simultaneously. Unreachable targets can occur when the goal is either beyond the reachable range of the kinematic chain or positioned in such a way that no combination of joint movements can orient the chain to reach it. On the other hand, there are instances where more than one solution exists (see Fig. 2.3). It is up to the inverse kinematics method to select the most suitable solution.

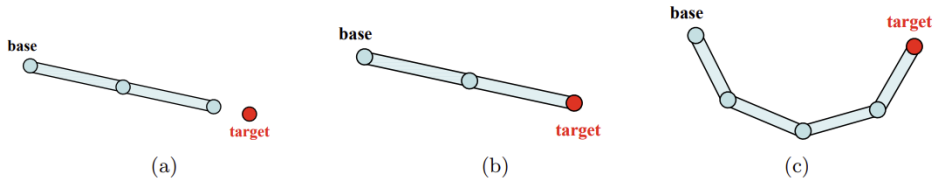

Figure 2.3 Possible solutions of the inverse kinematics problem: **(a)** the target is unreachable; in many cases it is impossible for the linked structure to touch the target, **(b)** one solution; there instances where there is only one solution to the problem, **(c)** many solution; most of the times the inverse kinematics problem has more than a single solution [16].

The Denavit-Hartenberg method [15], the product of exponentials method [14], the trigonometric method [17], and the dual quaternion method [18] are among the most widely used approaches for formulating the forward kinematics of serial chain manipulators. While the Denavit-Hartenberg method is consistent and the most concise of these, it does have certain limitations [19].

Various models have been developed to solve the inverse kinematics problem, drawing from diverse areas of study. The most common methods for solving inverse kinematics include:

- **Analytical methods**: exact solutions are sought, but they only work for simple robotic structures.
- **Numerical methods**: they are iterative algorithms which are used to find an approximate solution.

Given the limitations of analytical methods, the most common approaches, which will be presented in this state of the art, are all numerical. The most popular one involves use the Jacobian matrix to find a linear approximation for the inverse kinematics problem 14. The Jacobian solutions linearly model the end-effectors' movements relative to instantaneous system changes in link translation and joint angle. Several different methodologies have been presented for calculating or approximating the Jacobian inverse, such as the Jacobian Transpose [20], Damped Least Squares (DLS) [21], Damped Least Squares with Singular Value Decomposition (SVD-DLS) [22], Selectively Damped Least Squares (SDLS) [23] and several extensions. Jacobian inverse solutions produce smooth postures; however most of these approaches suffer from high computational cost, complex matrix calculations and singularity problems. The second family of inverse kinematics solvers is based on Newtonian methods. These algorithms aim to find target configurations by solving a minimization problem, resulting in smooth motion without erratic discontinuities. The most well known approach is the Broyden, Fletcher, Goldfarb and Shanno (BFGS) method [24]. However, the Newton methods are complex, difficult to implement and have high computational cost per iteration. A widely used inverse kinematics solver method is the Cyclic Coordinate Descent (CCD) algorithm [25]. CCD is a heuristic iterative method with low computational cost for each joint per iteration, which can solve the inverse kinematics problem without

matrix manipulations; thus it formulates a solution very quickly. However, CCD has some disadvantages; it can suffer from unrealistic animation, even if manipulator constraints have been added, and often produces motion with erratic discontinuities.

2.3 Grasp detection

In Section 2.1, grasp detection and its importance within a grasping system were introduced. Now, the most common techniques in the literature for performing this task will be explored in detail. As shown in Fig. 2.4 grasp detection techniques are mainly classified into *analytic* and *data-driven methods*.

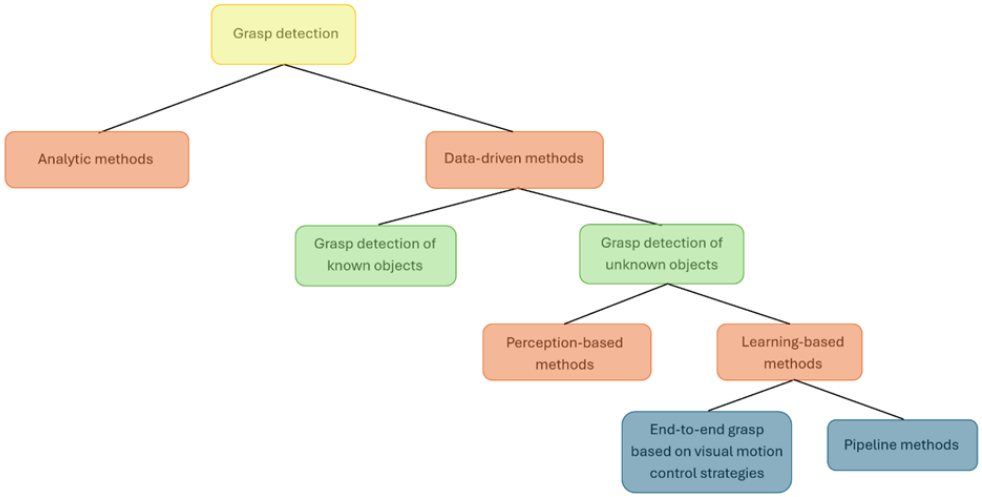

Figure 2.4 Taxonomy of grasp detection techniques.

The basic principle of analytical methods is to determine the appropriate grasp pose by analyzing the target object's geometry, motion state, and force based also on important concepts like *form-closure* and *force-closure*. Form-closure refers to the ability of a robot to completely restrain an object's motion in any direction, ensuring it remains stationary regardless of external forces, by configuring an appropriate grasping position [26]. Force-closure means that appropriate contact forces at the grasp points counteract external forces acting on the object, completely constraining its movement [27]. The detection quality resulting from these types of methods largely depends on the exact geometric model of the object and robotic hand. Furthermore, the grasp detection results based on analytic methods have poor adaptability in an unstructured environment and are unsuitable for real-time applications.

Data-driven methods can be applied to grasp detection for both *known* and *unknown objects*. Grasp detection methods for known objects require the target object to have a complete 3D model [28]. This kind of method has a simple principle, high detection accuracy, and easy implementation, but it has significant limitations in complex unstructured environments. Therefore, grasp detection methods for unknown objects have become a current research hotspot.

The classification of unknown object grasp detection techniques can be mainly

divided into *perception-based* and *learning-based* methods. Perception-based methods generate and evaluate grasp candidates by recognizing structures or features in image data, which are usually only for objects of specific shapes. However, the detection time and accuracy cannot be guaranteed, so their practical application is limited [29] [30]. Learning-based methods are the focus of current research in robotic grasping. Such methods are not limited by the complexity of the environment or the shape of the target objects, especially deep learning-based ones, which have greatly contributed to the development of unknown object grasp detection technology. They can be divided into two categories: *pipeline* methods and *end-to-end grasp based on visual motion control strategies*.

Pipeline methods include sliding window methods [31] and one-shot detection methods [32]. Sliding window methods have the advantages of simple structure, good detection accuracy, and certain generalization ability. The main limitation is that the optimal solution is obtained by traversal search, so the efficiency is low [33]. This has been improved by using one-shot detection methods, which mainly adopt ResNet-based network models and greatly reduce detection time while ensuring higher accuracy. Although one-shot detection methods do not require an iterative search, they still consume a lot of training time when the network model structure is complex and needs to be trained on large datasets. In this regard, researchers often use deep transfer learning techniques to employ pretrained deeper convolutional networks for this type of task [34].

End-to-end grasp based on visual motion control strategies, which do not require independent configuration for the planning control system, can directly realize the mapping from images to grasp actions [35].

2.4 Grasp planning

Grasp planning was introduced in Section 2.1 along with the other subsystems of the general framework of a robotic grasping system. It is an intermediate step between capturing information through the robot's sensors, aimed at identifying the target object and its position, and the actual grasping, which occurs in the final subsystem.

So far, in this state of the art review, we have focused on the most widely used techniques in robotics for grasp detection and the computation of direct and inverse kinematics, without specifying the types of robots involved. This is because these are general concepts that do not significantly change from one type of robot to another. However, in this case, grasp planning will be described from the specific perspective of humanoid robots, as their physical structure means that grasp planning does not only concern the arm but often involves the entire body. A humanoid robot often not only needs to plan the grasp but also approach the object using its wheels or lower limbs (something that, for example, an industrial robotic arm does not need to worry about, as it is fixed or mounted on a mobile base, and thus grasp planning is focused solely on the positioning of the arm).

Because humanoid robots are meant to operate in cluttered domestic environments, planning algorithms are needed to generate collision-free trajectories. However, planning a reaching or re-grasping motion requires choosing a feasible

grasping pose with respect to an object to manipulate and finding a configuration of the robot's joints that places the robot's end effectors at this pose. Thus, the planning algorithm must decide which of the feasible grasping poses should be selected and determine the robot's joint configuration for that pose [36]. A classical step-wise planning approach is depicted in Fig. 2.5.

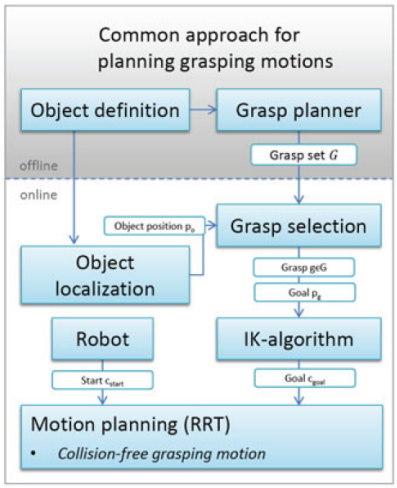

Figure 2.5 Planning grasping motions with a classical step-wise approach [36].

In this approch, in an offline phase, a grasp planner is used to pre-compute a set of potential grasping configurations. In the online phase, this set of grasps G is employed to select a feasible grasp $g \in G$ with respect to the localized object position. The target pose $p \in SE(3)$ is calculated by applying the object-related grasping pose g to the object's pose in workspace. By solving the inverse kinematics problem, a goal configuration c_{goal} can be computed, which is passed to the motion planning algorithm. Several problems may arise with such step-wise approaches. The selection of the grasp $g \in G$ is made without knowing whether g is reachable. Additionally, there is no guarantee that a collision-free grasping motion exists to move the end effector from the current position c_{Start} to c_{Goal}. Determining if such a solution exists is computationally as complex as solving the motion planning problem.

To overcome these problems, numerous alternative approaches have been proposed in the literature. The concept of eigengrasps enables grasp planning in a low-dimensional subspace of the actual hand configuration space, simplifying the complexity of high-dimensional hand configurations. Many works are based on this idea [37], [38], [39]. Approaches based on rapidly exploring random trees (RRTs) are widely utilized for planning grasping and reaching motions in humanoid robots, leveraging their efficiency in single-query path planning [40]. Another method involves planning grasping motions using pre-defined sets of grasping poses, where offline-calculated target poses are used to search for inverse kinematics solutions during planning [41]. However, these methods face limitations such as susceptibility to local minima in steepest descent optimization and insufficient consideration of orientation. To address these challenges, object-specific task maps have been pro-

posed to simultaneously plan collision-free reaching and grasping motions [42]. This approach employs analytic gradients to optimize motion costs and select grasps on the manifold of valid grasps.

2.5 Control

In the control subsystem described in Section 2.1, the solution to the inverse kinematics is determined based on the information provided by the previous subsystem. The robot is then controlled to perform the grasp according to the identified solution. In Section 2.2, we have already discussed what inverse kinematics is and the methods to calculate possible solutions. Next, the main control techniques will be examined.

Different techniques can be used to control a manipulator. The technique followed, as well as the way it is implemented, can have a significant influence on the performance of the manipulator and, consequently, on the possible range of applications. On the other hand, the mechanical design of the manipulator influences the type of control scheme used [14].

Regardless of the specific type of mechanical manipulator, it is worth noting that the task specification (end-effector motion and forces) is usually performed in the *operational space*, while the control actions (generalized forces of joint actuators) are performed in the *joint space*. This fact naturally leads to considering two types of general control schemes, namely a *joint space control scheme* and a *operational space control scheme* [43]. In both schemes, the control structure features closed loops to exploit the good characteristics offered by the feedback.

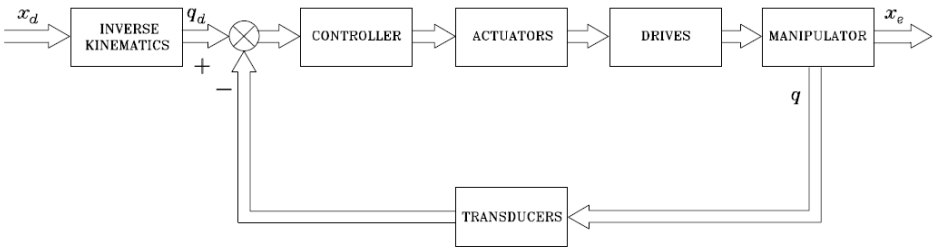

Figure 2.6 General scheme of joint space control [14].

As shown in Fig. 2.6 the joint space control problem can be divided into two subproblems. First, manipulator inverse kinematics is used to convert the desired motion requirements, x_d, from operational space into the corresponding joint space motion, q_d. Second, a joint space control scheme is designed to ensure that the actual joint motion, q, tracks the reference inputs. However, this approach has a limitation: the joint space control scheme does not directly affect the operational space variables, x_e, which are regulated in an open-loop manner through the manipulator's mechanical structure. Consequently, uncertainties in the structure—such as manufacturing tolerances, calibration errors, gear backlash, or elasticity—or inaccuracies in determining the end-effector's pose relative to the target object can reduce the precision of the operational space variables.

The operational space control approach adopts a global strategy that demands

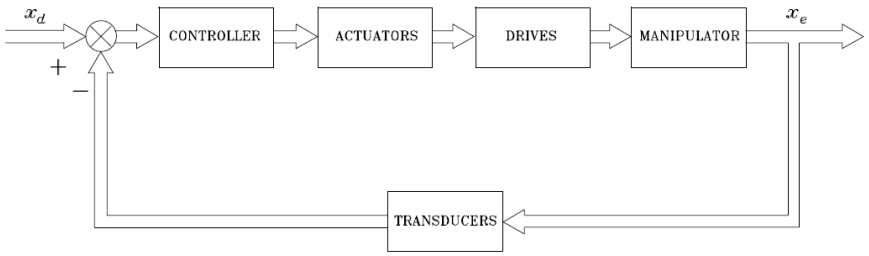

Figure 2.7 General scheme of operational space control [14].

increased algorithmic complexity, integrating inverse kinematics directly into the feedback control loop (Fig. 2.7). Its primary conceptual benefit lies in the ability to manipulate operational space variables directly. However, this advantage remains theoretical to some extent, as operational space variables are typically not measured directly but are instead derived by evaluating direct kinematics functions based on measured joint space variables.

2.6 Related work

So far, in this state of the art, we have generally examined the fundamental steps that constitute a robotic grasping system, analyzing, when necessary, how these processes are managed in humanoid robots. There is clearly a great variety of robots, characterized by different hardware and software specifications, as well as different physical and material properties. For this reason, we have chosen not to focus on specific robots, but to highlight exclusively the methodological aspect of the various approaches. Even in the description of humanoid robots and their characteristics, the emphasis has been on the shared properties of these robots, without delving into specific models. It is clear, however, that when conducting research, it is essential to explore in the literature how the task being worked on has been addressed and solved on the chosen reference platform or on similar platforms. The platform naturally shapes certain system design decisions, since the robot's hardware specifications or software constraints must be considered and managed.

As mentioned earlier, for this thesis' work the humanoid robot chosen as the platform for the system is the popular Pepper robot from Aldebaran Robotics (now SoftBank Robotics). This company has also produced two other humanoid robots, NAO and Romeo, all shown in Fig. 2.8, which share many similarities with Pepper in terms of their specifications. This section will briefly describe the approaches used to implement grasping systems on these robots. First, some basic information about the main functionalities of these robots will be provided. A more detailed analysis of Pepper's technical specifications will be presented in Section 3.1. All three robots use a similar operating system, called NAOqi, but they differ in some physical characteristics. NAO and Romeo are bipedal robots, while Pepper is equipped with wheels for movement. As for visual sensors, the three robots are very similar: in the most recent configurations, they are equipped with two front-facing 2D cameras with different fields of view. Additionally, Romeo and some versions of Pepper feature an ASUS Xtion 3D sensor, while stereoscopic vision is provided by a pair of 2D cameras

located behind the eyes (a feature absent in Romeo). In terms of degrees of freedom, Pepper has 5 degrees of freedom per arm, NAO has 6, and Romeo has 7.

(a) (b) (c)

Figure 2.8 **(a)** Pepper, **(b)** NAO, **(c)** Romeo.

Starting from Pepper, there are many studies demonstrating how Pepper has been successfully used as a companion or guide in various contexts, such as museum guides [44], elderly care [45], automated tours [46], service robots [47], education [48] [49], or promoting commercial products [50], leveraging its uniqueness to attract potential customers. However, given its nature, since it is designed to interact with people naturally but not to perform heavy-duty tasks or grasp objects precisely, and considering the conformation of its arms and hands, there have not been many specific studies on grasping task.

Two main works will be presented. The first study [51] addresses both autonomous navigation and grasping for the Pepper robot. The work is based on a Python script that allows Pepper to explore a room after creating a map and to perform the grasping of an object marked with Naomarks [52]. Naomarks are special visual markers designed to be recognized by the Aldebaran humanoid robots. They are similar to QR codes or Aruco markers but are specifically developed for the NAOqi robotics ecosystem. The second work [53], involves the design and implementation of a robust visual servoing framework for reaching and grasping behaviors with Pepper. It uses the 2D cameras provided by the robot and a marker-less model-based tracker to reach the target. Additionally, markers are used on the end manipulator to more accurately track the position of the hand in space.

Various grasping systems have been proposed that utilize the NAO robot as a platform. Compared to Pepper, NAO is more widely used in research due to its lower cost, making it a popular choice in robotics competitions. Three main works will be presented. The first research [54] integrates monocular stereo vision with kinematic

control to achieve grasping with NAO robot. The images are used to identify the target using a quantitative statistical algorithm, converting 2D image coordinates into 3D space via a pinhole perspective model. The robot's arm is controlled using direct and inverse kinematics to reach and grasp the target, guided by continuous visual feedback. Additional sound-based feedback helps the robot locate and move toward the user post-grasping. The second paper [55] proposes a method to enhance the NAO robot's ability to localize and grasp objects using the YOLOv8 network for target detection combined with monocular ranging for distance estimation. To reduce errors in long-distance monocular vision, a visual distance error compensation model is introduced. The approach also includes a grasping control strategy based on pose interpolation. The third work [56] uses 3D models to identify known objects. The system integrates a path planner to generate feasible motion trajectories.

Romeo is a more complex and expensive robot to produce, which has limited its widespread adoption. As a result, research projects using it to develop grasping systems are much more limited compared to the other two humanoid robots from Aldebaran. The paper [57], similarly to the work [53], explores visual servoing techniques to perform one- and two-handed manipulation tasks, including object grasping and solving a ball-in-maze game. It uses gaze control to maintain visibility of the hand and object, employs a master/slave control for dual-arm tasks and introduces methods to avoid joint limits. The 6D pose of both the hand and the target object is estimated using visual markers placed on them.

Chapter 3

Platform

In this chapter, the platforms used in this work will be described. Section 3.1 discusses the hardware and software specifications of Pepper. Section 3.2 introduces the QiBullet simulator, which was used to design and implement the proposed grasping system. Finally, Section 3.3 provides an overview of the Robot Operating System (ROS) framework.

3.1 Pepper

Pepper is a humanoid robot that is 1.2 meter tall and weighs 28 kilograms. It has 20 degrees of freedom in total, 17 distributed throughout the body and 3 in the base (Fig. 3.1). The latter is omnidirectional, allowing for holonomic planning and control of movement. Due to the long life of its battery, it can work for 12 hours comfortably and be charged if needed.

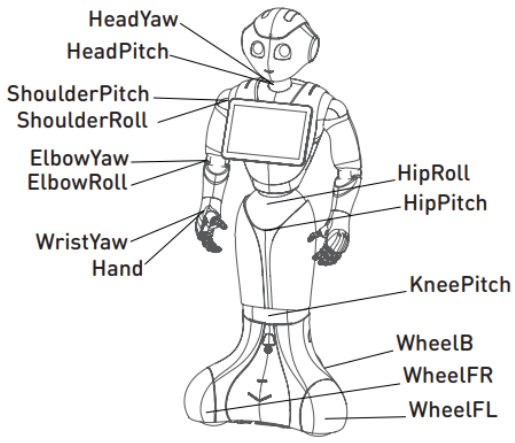

Figure 3.1 Pepper's degrees of freedom [58].

The robot is equipped with an IMU, made of a 3-axis gyrometer and a 3-axis accelerometer. To avoid obstacles, it uses two sonars, two infrared sensors and six lasers, three of which are pointing downwards and three dedicated to identifying objects in the area around the robot. It also has three bumpers near the wheels,

which are used to stop the robot in the event of a collision. To interact with users, Pepper is equipped with a system of four microphones located on the top of the head, two speakers and three tactile sensors: one on the back of each hand and one on the top of the head (see Fig. 3.2).

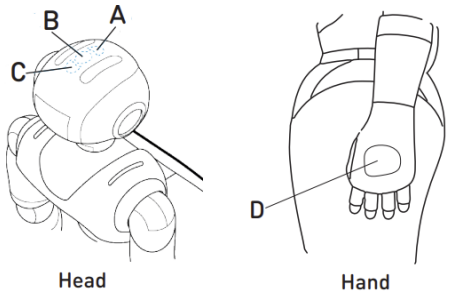

Figure 3.2 Pepper's tactile sensors [58].

It also has three groups of LEDs for non-verbal communication, located respectively in the eyes, on the shoulders and around the speakers in the ears.

The tablet on Pepper's chest is an Android tablet, which can be used to develop apps that integrate with the robot or as a display for web pages, pictures, or videos. Both the robot and the tablet have independent wireless connectivity.

As for visual perception, as already mentioned in Section 2.6, Pepper is equipped with two cameras with a native resolution of 640x480. One camera is positioned in the forehead, pointing slightly upward, and the other is in the mouth, pointing downward, as shown in Fig. 3.3. In the 1.8a version of the robot, there is an ASUS Xtion 3D sensor behind the eyes. In version 1.8, the one available at VisionLab, the robot uses a pair of 2D stereo cameras, also located behind the eyes, to obtain stereoscopic images. Version 1.8a of the robot was released before version 1.8. The shift from the 3D sensor to stereo cameras in the subsequent version is likely due to the fact that the Xtion camera appeared to provide erroneous depth maps, which also resulted in incorrect point clouds, as stated in this paper [59].

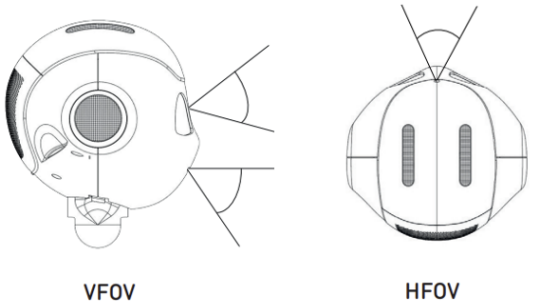

Figure 3.3 The vertical and horizontal fields of view of the top and bottom 2D cameras of the robot [58].

Computationally, Pepper is equipped with a 1.9 GHz Atom quad-core processor and 4 GB of RAM. It can run two operating systems: NAOqi 2.5 and NAOqi 2.9

[60], [61]. Both are hardware independent and their choice does not affect battery life. NAOqi 2.5 is a Linux-based operating system and supports SDK development in Python [62] and C++ [63]. NAOqi 2.9, on the other hand, is based on Android and uses Java and Kotlin as its main languages and allows development through the QiSDK library [64]. It is technically possible to upgrade from NAOqi 2.5 to NAOqi 2.9. However, it is not possible to downgrade the other way.

3.2 QiBullet simulator

Presently, Pepper and NAO models are available in simulation tools such as Gazebo [65], V-REP [66], Webots [67] and Choregraphe [68]. However, these implementations either lack the ability to accurately handle the physics of the model or to simulate complex environments. For this reason, QiBullet (Fig. 3.4) was chosen as the simulator for this work. Developed by SoftBank Robotics, QiBullet is designed to model the movements of the three humanoid robots produced by the company. This simulator does not handle dialogues or the Pepper's tablet.

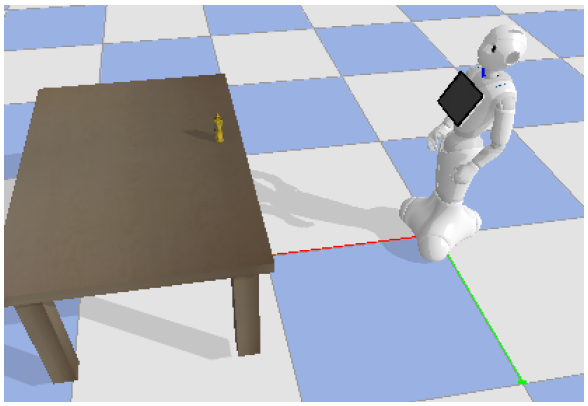

Figure 3.4 Pepper in the QiBullet simulator.

The QiBullet simulator is built on the Bullet physics engine [69] and the PyBullet module [70]. From these two frameworks, it inherits cross-platform capabilities, allowing the simulation tool to run on Linux, Windows, and macOS. The Bullet physics engine enables the simulation of physical properties such as gravity and friction. Additionally, QiBullet enables the integration of PyBullet APIs into your code, providing access to a wide range of robotics tools.

A Unified Robot Description Format (URDF) file defines the virtual robot model, detailing its links, masses, inertia matrices, and connecting joints. Mesh files linked to each component allow the engine to render the robot's visual appearance and perform collision detection.

The Python-based QiBullet API [71], layered over the PyBullet API, enables users to interact with the simulated robot and its environment. To facilitate integration into existing projects and maintain code consistency, the QiBullet API partially replicates the NAOqi API from the Python SDK, making interactions with virtual and real robots as seamless as possible. For machine learning applications, the API supports instantiating, resetting, and stopping multiple independent simulation

instances in parallel. It also allows users to spawn or remove virtual robots within an instance and adjust the position of the simulation's light source.

QiBullet provides a wide range of functions for testing Pepper's movements, such as positioning the robot in specific postures or moving it along defined axes. The robot's joints can be controlled individually or in groups to achieve target positions at specified speeds (Fig. 3.5). Additionally, the base of the Pepper virtual model can be controlled either by position or velocity.

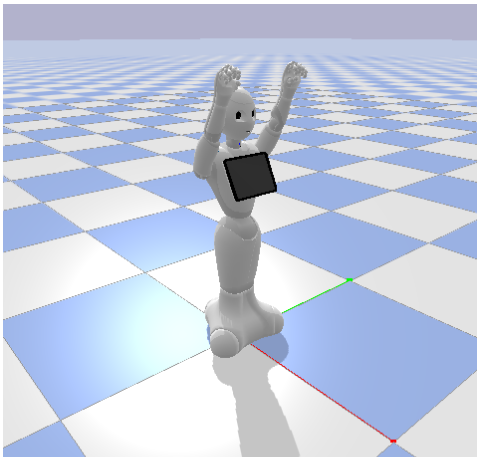

Figure 3.5 Pepper's arm joints are controlled to raise its arms.

The simulator also offers camera previews, providing insight into how the robot perceives its surroundings. Each Pepper and NAO virtual model includes two RGB cameras (see Fig. 3.6), while the Pepper model additionally features a depth camera. However, stereoscopic vision, introduced in newer NAO and Pepper physical versions, cannot be simulated. Like the NAOqi API, QiBullet allows users to select the resolution of synthetic images. The simulated camera parameters are calibrated to match those of the real robots, though real images from Pepper tend to be noisier than their synthetic counterparts.

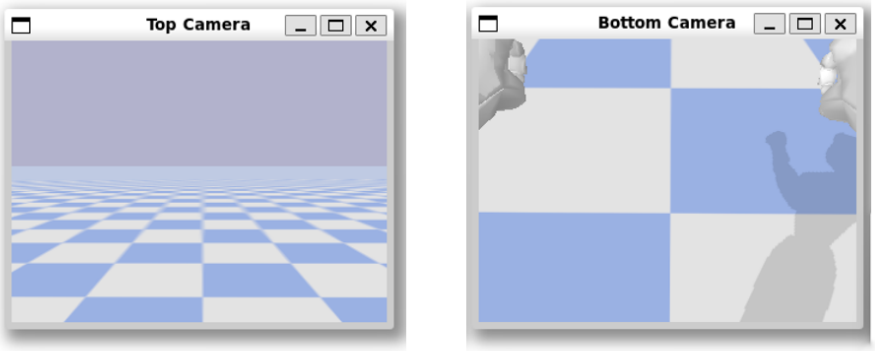

Figure 3.6 View from the top and bottom cameras of the Pepper robot while framing the same scene.

The Pepper virtual model also includes laser sensors at its base, replicating the real robot's hardware (Fig. 3.7). QiBullet supports the addition of lights to the simulation, enabling the creation of realistic environments with varied lighting conditions. Complex environments can be built by loading 3D objects via URDF, SDF, STL, or OBJ files, allowing the simulation of diverse scenarios the robot might encounter.

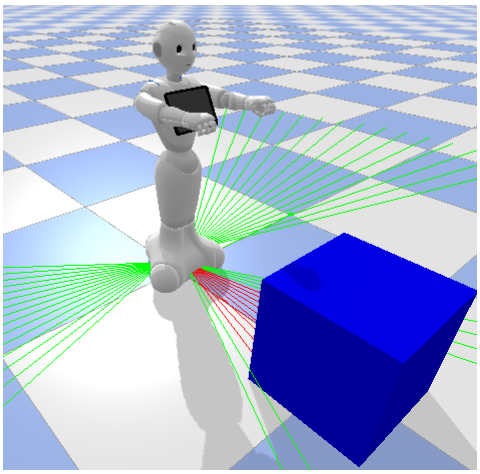

Figure 3.7 The robot's lasers detect the presence of an object.

3.3 Robot Operating System (ROS)

The Robot Operating System (ROS) [72] is an open source middleware framework with libraries and tools for robot software development. It includes state-of-the-art algorithms, an inter-process communication framework and visualization tools. It is used on many robots and by many research groups because of its hardware abstraction and package management, with many algorithms for perception, planning, and localization. ROS avoids the constant reinvention of foundational elements, providing standardized functionalities, reducing research and development time and costs, speeding up the "time to market" and enabling the integration of diverse expertise, from hardware management to artificial intelligence. Its architecture allows multiple components to communicate directly, synchronously or asynchronously, and supports languages such as Python, C++ and Lisp.

One of the key strengths of ROS is its community, which contributes to a growing ecosystem of packages and tools. ROS re-uses code from numerous other open-source projects. In each case, ROS is used only to expose various configuration options and to route data into and out of the respective software, with as little wrapping or patching as possible. To benefit from the continual community improvements, the ROS build system can automatically update source code from external repositories, apply patches, and so on.

ROS community also supports a ROS Interface node to bridge ROS and the NAOqi system but only NAOqi 2.5 is compatible directly with ROS.

Chapter 4

Proposed approach

In Section 4.1, the limitations of the platforms used, which inevitably influenced the choices made in the design and implementation of the system are presented. In Section 4.2, the architecture and methodological functioning of the system are described. In Section 4.3, the differences and similarities between the APIs provided by the simulator to perform the various tasks required in this system—such as vision and motion—and those offered by the Python SDK used to program the physical robot are outlined. In Sections 4.4, 4.5, 4.6, 4.7 and 4.8, the methodological and implementation choices adopted for each of the previously described modules are illustrated in detail. In Subsections 4.4.1, 4.5.1 and 4.8.1, the theoretical characteristics of the models introduced in their respective sections are described in detail, explaining how they function.

4.1 Platform constraints

As we saw in Chapter 2, there are many choices to make when designing a grasping system. Among them, it is important to decide whether you want the robot to grasp only known objects or also unknown objects and how you want to describe the object you want to manipulate. It is also essential to evaluate the advantages and disadvantages of the platform before moving on to the implementation. For example, it is important to identify which available sensors can be useful to achieve your goal and what their limitations are. In terms of hardware and software, the grasping task with the Pepper robot has the following constraints:

- Performing force-closure single-handed grasps is not possible with Pepper, since it is not able to move its fingers individually. The hand can only be fully open or fully closed.
- Pepper does not have advanced tactile sensors on its hands. It only has capacitive sensors, but these are more useful for detecting human touch than for providing true tactile feedback on objects. Additionally, these sensors are only present on the top of each hand, making them not very useful for the grasping task.
- Pepper's 2D cameras are limited. They have a frame rate of 15 frames per

second (fps) and operate at 10 Hz. This can affect object recognition and tracking during deployment.

- Pepper can only lift 200 grams. Therefore, the target object should be as light as possible and must fit within Pepper's hands, which are relatively small.
- The arms have 5 degrees of freedom (DOF). As a result, there are some positions and orientations that the robot cannot reach.
- Only one camera can be used on the robot at a time. For example, if you use the bottom camera, you must turn off the top camera and vice versa.
- The simulator only offers the possibility to simulate the top and bottom cameras of the robot. It is not possible to simulate stereoscopic cameras.

4.2 System architecture

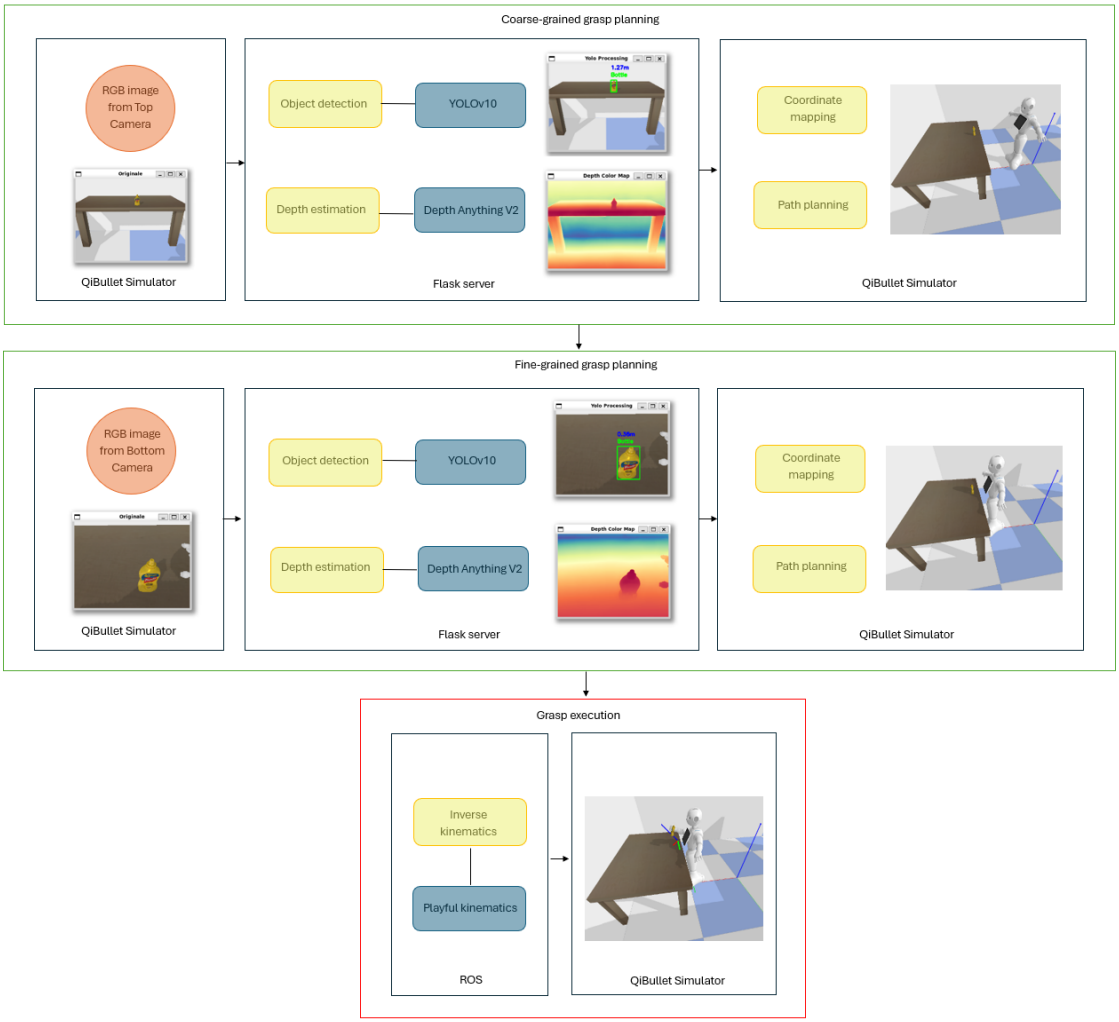

Figure 4.1 System architecture.

As shown in Fig. 4.1, the system uses 5 key modules:

- **Object detection**: identifies the target object in the images captured by the robot's cameras. This is a very important step because we want to grasp an unknown object, so we need to identify it first. The task consists of both detecting where the target object is and labeling it. Specifically, rectangular boxes are placed around the detected objects.
- **Depth estimation**: estimates the distance of each point in an image from the camera, creating a depth map. In general, depth estimation algorithms can be divided into monocular or multi-view methods, depending on the number of images required to infer the depth. In this case is used a monocular approach.
- **Coordinate mapping**: transforms the detected image plane coordinates into the robot's base coordinate system.
- **Path planning**: generates a feasible path from the manipulator to the target object, optimizing the orientation, distance and position of the robot with respect to the object to obtain the best possible grasping pose.
- **Inverse kinematics**: calculates the joint configurations needed for the robot to reach a given end effector position and orientation to finally perform the grasp.

The system sees the communication of three main components:

- **QiBullet Simulator**
- **Flask Server**
- **ROS**

All the modules of the system were written in Python and run on a laptop with an Intel Core i9 processor and an Nvidia RTX 4060 GPU. The specific version of ROS installed is ROS Melodic. The vision modules (object detection and depth estimation) are executed on a Flask server running on Windows, while QiBullet and ROS were installed on Ubuntu 18.04.6 LTS, running via Windows Subsystem for Linux (WSL), a Windows feature that allows you to use a Linux environment directly on a Windows operating system, without the need for a virtual machine or dual boot.

The simulator allows you to plan movements according to the physical laws of a real environment, such as gravity and friction. The virtual model of the robot is created by loading a URDF file that contains information about Pepper's links, masses, inertia matrices and connection joints. Using additional URDF files, an indoor office-like environment is reconstructed, with walls and a table (see Fig. 4.2). The chosen target object is a small bottle, as it is both compact enough to fit in Pepper's hand and light enough to be lifted without exceeding the robot's physical constraints. Additionally, the geometry of this type of object lacks irregular shapes or moving parts, making the grip more stable.

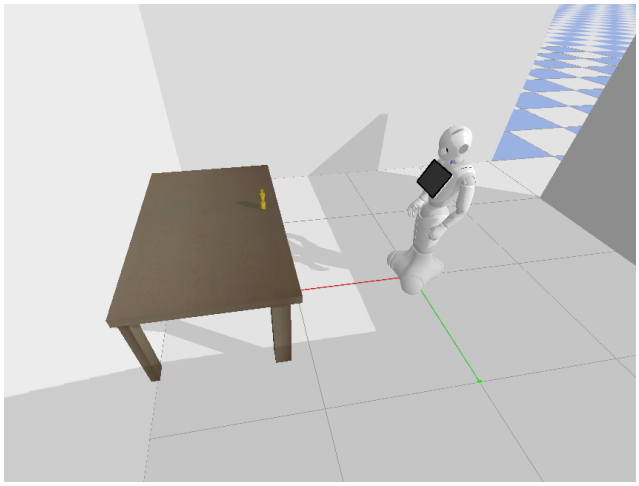

Figure 4.2 The office-like environment reconstructed in QiBullet.

Since it is not possible to simulate stereoscopic cameras, only the two simulated RGB cameras were used throughout this work. As the robot rotates on its axis to detect the presence of the target object, the top camera captures images at a frame rate of 15 fps. These images are then sent to the Flask server, which runs the object detection and depth estimation modules. As shown in Fig. 4.3, the top camera offers a wide field of view, making it ideal for orienting the robot and guiding its navigation in space.

Figure 4.3 Top camera view.

Once the images are processed, they are returned to the simulator, providing information regarding the position of the target object in space, necessary for the execution of the coordinate mapping and path planning modules. As the robot approaches the table and the bottle, a more detailed grasp planning is performed using the images captured by the bottom camera. This camera, pointing downward, offers a better perspective on surfaces close to the robot, making it ideal for obtaining a more accurate assessment of the distance from the target object (Fig. 4.4).

Figure 4.4 Bottom camera view.

These images are sent back to the server, which performs object detection and depth estimation again, providing a more accurate depth estimate. At this point, a new step in the path planning algorithm adjusts the robot's position slightly to optimize its grasping pose. Subsequently, the grasping of the object is carried out through inverse kinematics, implemented using ROS.

4.3 Differences and similarities between QiBullet and NAOqi APIs

To develop this system, APIs primarily related to vision and movement were utilized. As described in section 3.2, the QiBullet API partially replicates the NAOqi API from the Python SDK.

A key concept in the NAOqi framework is modules. In the Python SDK, modules are essentially predefined collections of functions and classes that enable interaction with specific features or capabilities of the Pepper robot. Each module typically corresponds to a particular aspect of the robot's functionality, such as movement, speech, vision, or interaction with the environment. For example, there is a module to control the robot's arms, another to manage its speech output, and yet another to process sensor data. The APIs provided by the Python SDK allow interaction with the robot's features through these modules.

As previously mentioned, QiBullet does not handle functionalities related to the robot's interaction with humans, such as speech, or the playback of photos, videos, or audio on the tablet; it focuses solely on aspects concerning the robot's movement. For this reason, the QiBullet APIs replicate only the most significant functionalities tied to the vision and movement modules. Specifically, in this work, the QiBullet APIs used replicate the functionalities of the ALMotion, ALRobotPosture, and ALVideoDevice modules. The ALMotion module provides methods that facilitate robot movement, allowing control over joint stiffness, joint position, and navigation from one point to another. The ALRobotPosture module enables the robot to adopt various predefined postures, with the option to adjust the speed at which these postures are applied. The ALVideoDevice module is responsible for capturing images from the video source (i.e., the robot's cameras).

Fig. 4.5 provides a detailed overview of the QiBullet APIs used and their equivalents in NAOqi.

Functionality	QiBullet API	NAOqi API
Base movement	`pepper.moveTo`	`motion_service.moveTo`
Joint control and hand control	`pepper.setAngles`	`motion_service.setAngles` or `angleInterpolationWithSpeed`
Posture Management	`pepper.goToPosture`	`posture_service.goToPosture`
Camera Management	`pepper.subscribeCamera`	`video_service.subscribeCamera`
Reference Frame Handling	Uses the PyBullet APIs	Uses `motion_service.getPosition`

Figure 4.5 Differences between QiBullet and NAOqi APIs.

As we can observe, the most significant differences lie in how the APIs are called. In NAOqi, it is necessary to register to a service, that is, an instance of a module made available by the NAOqi Service Manager and associated with a session that must be created every time a connection to the physical robot is established. In the simulator, this is not required; it is sufficient to create a virtual robot using the simulation manager. In Fig. 4.5, `Pepper` refers to the virtual robot, while `motion_service`, `posture_service`, and `video_service` represent registrations to the requested services.

The parameters that should be passed to each method are quite similar between the two APIs. Specifically, `moveTo` in both cases requires the distance along the X-axis in meters, the distance along the Y-axis in meters and the rotation around the Z-axis in radians. `setAngles`, on the other hand, requires the name of the joints or chains you want to control, the angle values in radians and the fraction of maximum speed to use.

QiBullet does not support `angleInterpolationWithSpeed`, which is instead supported by NAOqi and is used when smooth, controlled movements are needed or when executing complex sequences of movements. `angleInterpolationWithSpeed` takes the same parameters as `setAngles` but uses an interpolation curve to gradually transition from the current joint positions to the target positions. `goToPosture`, for both the APIs, requires the name of the predefined posture to be reached and the speed at which it should be achieved.

For camera usage, it is necessary to specify the camera index to be used (0 for the top camera, 1 for the bottom camera, and 3 for the stereo cameras, though the latter are only usable with NAOqi), the resolution at which images should be captured, the desired color space and the frames per second.

Finally, NAOqi provides built-in functions to obtain the position relative to a reference frame with the method `getPosition`. QiBullet, on the other hand, relies on the QiBullet APIs for these functions.

4.4 Object detection module

As mentioned, the target object to detect is a small bottle. The bottle model used in the simulator is a 3D model in URDF format, taken from one of the objects present in the well-known YCB dataset (Yale-CMU-Berkeley Object and Model Set) [73], a standardized set of physical objects used in research on robotics, computer vision and robotic manipulation (see Fig. 4.6). This set includes everyday objects with different shapes, sizes, textures, weights and stiffnesses.

In the future, other objects will be tested as target objects.

Figure 4.6 The 21 reconstructed object models of the YCB-Video dataset. The used bottle has been highlighted.

Pepper's processor is not suitable for performing computationally expensive operations such as object detection and depth estimation. Pepper uses a processor from the Intel Atom family, designed to consume low power and handle light operations, such as motor control, basic sensor processing, and managing voice and visual interactions. It also does not have a dedicated GPU. For these reasons, the chosen approach is to use an HTTP server, allowing the use of a powerful GPU that can process images captured by Pepper in real-time and send the results back to the robot for the next action. The server was built using Flask, a lightweight Python web framework that handles HTTP requests (such as GET, POST, PUT, DELETE) and can return responses to the client. This solution enables the execution of heavy but more accurate models on a computer rather than directly on the robot, resulting in much faster and more precise processing of the captured images.

The model chosen to perform object detection is YOLOv10. One of the reasons YOLO was selected is that it is already trained to detect bottles, as well as a wide range of other objects such as umbrellas, soccer balls, cups, glasses, forks, knives, spoons, chairs, as well as people, animals, fruits, foods and electronic devices. In Fig. 4.7, YOLO is utilized in the simulator to detect objects such as a book, a bowl and a bottle.

In particular, YOLOv10 in its small version was chosen because it was designed for applications requiring a good balance between speed and accuracy, such as robotics and is optimized for use with a GPU. To further improve the aspect related to image processing speed, images with a resolution of 320x240 were used.

Figure 4.7 Object detection with YOLOv10 in QiBullet.

Using OpenCV, the most popular and widely used open-source library for computer vision, after detecting the bottles in the scene, bounding boxes were drawn to highlight their location within the image, as shown in Fig. 4.7 and 4.8.

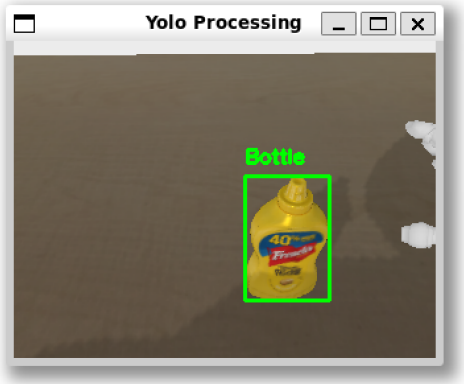

Figure 4.8 Object detection with YOLOv10 on images captured with the bottom camera.

In Section 4.4.1, the architecture of YOLO, the model that revolutionized the field of object detection, will be described in detail.

4.4.1 YOLO (You Only Look Once)

The YOLO (You Only Look Once) [74] family of models is a set of convolutional neural networks (CNNs) designed for real-time object detection. With the advent of neural networks, researchers have achieved new milestones in the field of object detection systems. Neural networks were inspired by the workings of the human brain and learn complex nonlinear mappings between inputs and outputs. Deep learning architectures are extensions of neural networks, characterized by a greater number of layers.

A CNN [75] is composed of a series of specialized layers that allow it to extract meaningful features from an image (Fig. 4.9). Convolutional layers apply filters, or kernels, to small portions of the image to detect local patterns such as edges and textures. Each filter generates a feature map that highlights specific image features. Pooling layers reduce the size of the feature maps while preserving the most relevant information; for example, max pooling selects the maximum value in a given region. This process makes the network more efficient and robust to image variations. Finally, the fully connected layers, after feature extraction, pass the data to dense layers to make final decisions, such as object classification.

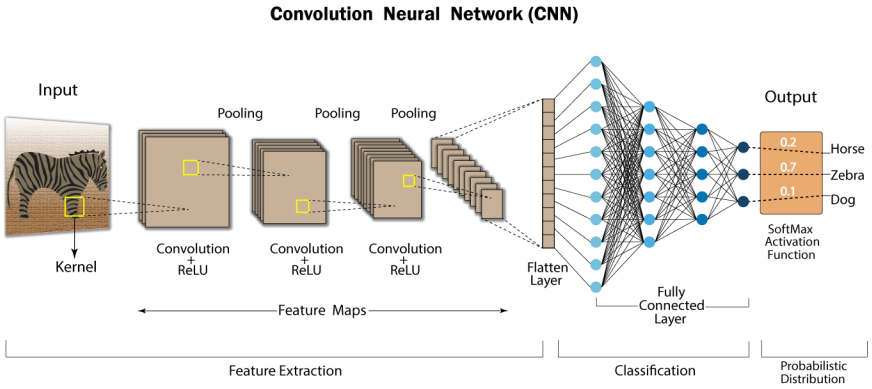

Figure 4.9 The architecture of a convolutional neural network (CNN).

CNNs are particularly effective for object detection for two main reasons. First, they detect hierarchical patterns: the first layers are able to recognize edges and textures, while deeper layers recognize parts of objects and, finally, entire objects. Second, CNNs are unaffected by translations and scale changes, thanks to the use of pooling, which allows them to recognize an object regardless of its position and size.

R-CNNs (Region-based Convolutional Neural Networks) [76] are an evolution of CNNs specifically designed for the task of object detection. They are among the first models to combine CNNs with region proposal techniques, enabling the localization and classification of multiple objects within an image. R-CNN consists of three steps or modules. The first step searches for different regions within the image that could potentially contain objects. The second step uses a large convolutional neural network (CNN) [77] to extract features from these regions. Finally, a support vector machine (SVM) [78] predicts whether an object exists within each region and classifies it.

Single-Stage Neural Networks as opposed to region proposal-based methods, single-stage neural networks use just a single network for both object localization and labeling. These methods are fast and efficient for real-time use. The most popular network of this type is YOLO, a fast algorithm that uses a single neural network to find regions within the image that may contain an object and to predict bounding boxes around possible objects and assign each a probability value.

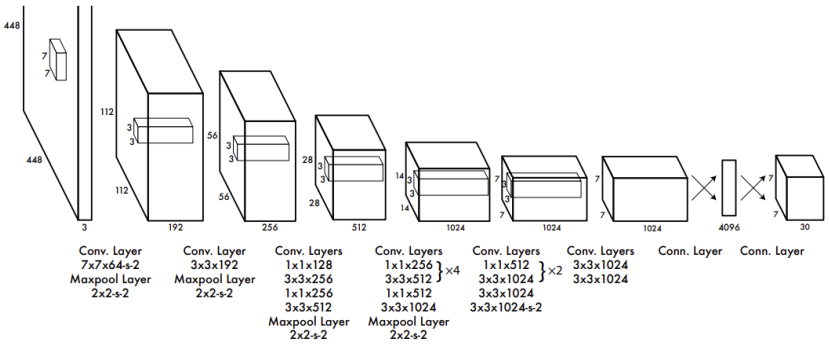

Figure 4.10 The YOLO architecture [74].

The YOLO model is made up of three key components (Fig. 4.10): the head, neck, and backbone. The backbone is the part of the network made up of convolutional layers to detect key features of an image and process them. The backbone is first trained on a classification dataset, and typically trained at a lower resolution than the final detection model, as detection requires finer details than classification. The neck uses the features from the convolution layers in the backbone with fully connected layers to make predictions on probabilities and bounding box coordinates. The head is the final output layer of the network which can be interchanged with other layers with the same input shape for transfer learning.

4.5 Depth estimation module

As previously mentioned, stereo cameras cannot be simulated in QiBullet, so a monocular approach was chosen to perform depth estimation. Monocular depth estimation (MDE) is a way to determine how far away things are in a picture taken with just one camera. Unlike stereoscopic techniques, which rely on multiple viewpoints to infer depth, monocular depth perception algorithms must extract depth cues from various image features such as texture gradients, object sizes, shading and perspective. The challenge lies in translating these inherently ambiguous cues into accurate depth maps, which has seen significant advancements with the advent of deep learning. The theoretical foundation of monocular depth perception is rooted in the understanding of how humans perceive depth with a single eye. Psychological studies suggest that the human visual system utilizes a series of cues, including linear perspective, texture gradient and motion parallax, to gauge depth. Leveraging these insights, computer vision researchers have developed algorithms that mimic this capability, using patterns and inconsistencies within a single image to estimate distances.

The MDE model selected for this work is the state-of-the-art Depth Anything model. At the beginning of this thesis, only the first version of this model was available [79], so it was initially integrated into the system. Upon the release of version 2 [80], the system was adapted to incorporate this updated model (Fig. 4.11). In Section 4.5.1, a detailed comparison of the architectures of the two versions will be provided, while in Section 5.2, their performance within the proposed system will be compared.

The URDF model of the bottle used in the simulator represents a non-transparent plastic bottle. This choice was made to address the difficulties encountered by depth estimation models in accurately detecting the depth of transparent objects. This issue was particularly evident with Depth Anything v1 and is generally due to the fact that these models rely on visual cues such as texture, edges, and color variations to estimate an object's distance. Transparent surfaces, which often lack well-defined textures, are frequently challenging to detect. Additionally, light refraction and reflection pose significant obstacles: as light passes through the object, its path is altered, distorting the perceived image, while environmental reflections can be mistaken for the background rather than recognized as part of an object at a specific depth. The problem is further compounded by the scarcity of adequate training data. Most datasets used for these models lack examples of transparent objects with precise annotations, causing the system to either ignore them or misjudge their depth.

The depth is extracted by estimating the central point of the bottle from the bounding box. This is because a bottle, when considering only its central part, can be approximated as a convex object, and its central point is indicative of the entire surface. The central point of a convex object is also the one that remains most visible and least prone to errors due to occlusions or reflections. If the depth were taken from an edge, errors could arise due to effects such as the loss of contour information.

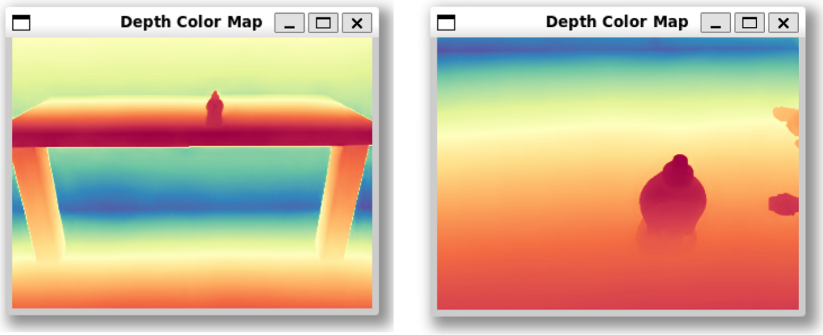

Figure 4.11 Depth color maps obtained with Depth Anything v2 in QiBullet simulator.

In partcular, what we aim to obtain from Depth Anything is a metric depth estimation. The metric estimation of Depth Anything is based on an adaptation of the pre-trained model with an additional fine-tuning step on metric datasets. There is a key parameter, called `max_depth`, which is fundamental to obtain the numerical distance value that allows the model to be adapted to the desired metric scale and specific environment (indoor or outdoor). If the estimation is to be performed in

an indoor environment, the developers of Depth Anything [81] recommend setting `max_depth = 20`, limiting the estimates to 20 meters, ensuring that the output is consistent with the typical dimensions of an indoor space. With max_depth = 80, the model would adapt to larger outdoor scenarios. With `max_depth = 20`, the resulting depth map will have realistic values, such as 1.5 meters for a nearby chair or 10 meters for a distant wall, but never exceeding 20 meters.

Using OpenCV, after performing the metric depth estimation, the distance in meters is displayed above the target object, as shown in Fig. 4.12.

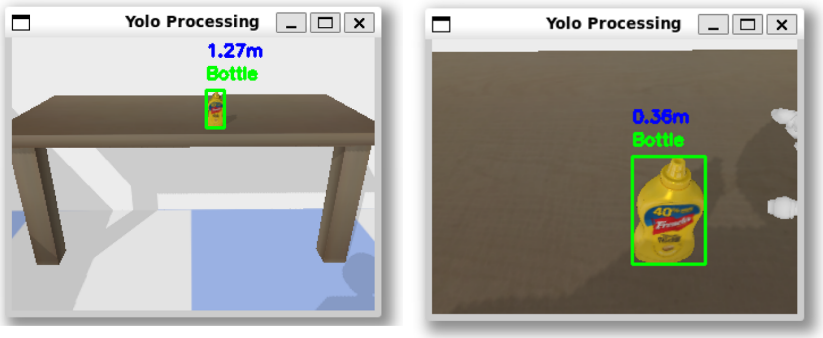

Figure 4.12 The distance estimation in meters is displayed after being calculated by Depth Anything v2.

4.5.1 Depth Anything V1 and Depth Anything V2

In order to effectively utilize a large amount of relatively cheap unlabeled data to improve learning performance, researchers have developed semi-supervised learning methods for monocular depth estimation. These approaches incorporate additional information, such as synthetic data or surface normals, to reduce the model's reliance on ground truth depth maps. This enhances scale consistency and improves the accuracy of estimated depth maps. Synthetic data, generated by graphics engines, offers a practical way to gather vast amounts of depth data, though bridging the gap between synthetic and real-world data remains a key challenge during training [82].

Depth Anything and its updated counterpart, Depth Anything V2, are among the latest and most advanced semi-supervised techniques for monocular depth estimation. By automatically annotating nearly 62 million unlabeled images, Depth Anything vastly expands the training dataset, enabling the model to learn from a much wider variety of scenes and lighting conditions than previously possible. It achieves this through a data engine that collects and annotates massive quantities of unlabeled images, significantly increasing data coverage, a critical factor in reducing generalization errors. The methodology relies on two core strategies: using data augmentation tools to create more challenging optimization targets and introducing auxiliary supervision to ensure the model inherits rich semantic priors from pre-trained encoders. Together, these techniques push the model to actively acquire additional visual knowledge and develop robust representations.

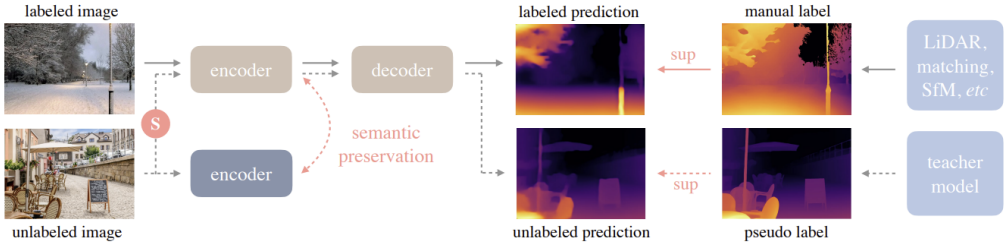

Figure 4.13 The Depth Anything pipeline. Solid line: flow of labeled images, dotted line: unlabeled images. [79]

For labeled images, the process is straightforward: these images undergo feature alignment, leveraging manual labels and supplementary techniques like LiDAR matching [83] or Structure from Motion (SfM) [84] to produce labeled predictions. This pathway follows traditional supervised learning principles, where this first part of the model —referred to as the teacher model—learns directly from explicitly labeled data (see Fig. 4.13).

For unlabeled images, the approach is more innovative, harnessing large-scale unlabeled data to enhance the model. Unlabeled images are enhanced with strong perturbations (denoted as "S" in the diagram in Fig. 4.13), which are designed to challenge the student model. These perturbed images are then processed through the same encoder-decoder structure as labeled images. However, instead of relying on manual labels, they leverage pseudo labels generated by the teacher model.

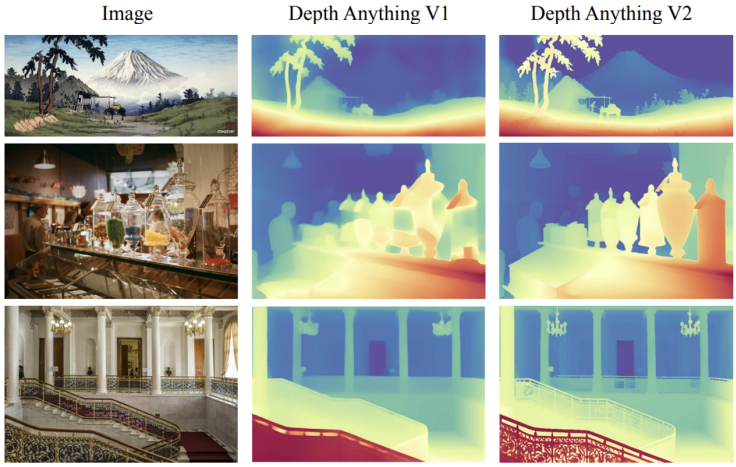

Figure 4.14 Comparison between Depth Anything V1 and Depth Anything V2 in open-world images [80].

While Depth Anything delivers impressive results, it struggles with transparent objects and fine-grained details (Fig. 4.14). The authors of Depth Anything V2 suggest that the biggest bottleneck for model performance isn't the architecture itself, but the quality of the data. Most labeled datasets, derived from sensors, tend to be noisy, overlook intricate details, produce low-resolution depth maps, and falter under varying lighting conditions or with reflective and transparent objects. To address this, Depth Anything V2 shifts away from real-world sensor data entirely,

training the teacher model exclusively on synthetic images generated by graphics engines. The teacher model then annotates millions of unlabeled real-world images, and the student model is trained solely on these pseudo-labeled datasets. This approach markedly improves performance by leveraging cleaner, more consistent synthetic data as a foundation.

4.6 Coordinate mapping module

When the bottle is detected in an image taken by a camera, its coordinates are initially expressed in the reference frame of the camera that acquired the image. It is necessary to calculate the bottle's position relative to the robot's base so that the robot can grasp it. This calculation requires several steps.

First, the position and orientation of the camera with respect to the robot base are retrieved. In the URDF file, necessary to load the virtual robot into the simulator, it is possible to obtain a series of information regarding its links and joints. In this type of file, the cameras are modeled as links since they are physical objects with a specific position and orientation relative to the robot and they are integrated into it by being connected to links. As previously mentioned, QiBullet inherits the methods available in PyBullet. Among them there is `getLinkState`, a function that returns the state of a given link, including information such as its position and orientation.

Once this first piece of information is obtained, the second step is to calculate the position of the bottle with respect to the camera. To do this, it is necessary to convert the center of the bounding box – which corresponds approximately to the center of the bottle – from the 2D format of the acquired image to a 3D format. For this purpose, the focal length is used. Let us consider a three-dimensional Cartesian reference frame, in which the x-axis represents the horizontal coordinate, the y-axis the vertical coordinate, and the z-axis the depth, perpendicular to the plane formed by x and y. To convert the 2D coordinates (x, y) of an image into 3D coordinates using the focal length f, it is necessary to consider perspective and the relationship between the image plane and the 3D space, which can be expressed as:

$$x = f \cdot \frac{X}{Z} \tag{4.1}$$

$$y = f \cdot \frac{Y}{Z} \tag{4.2}$$

where (X, Y, Z) are the coordinates in 3D space. The value of Z has been calculated using Depth Anything v2, so to obtain X and Y, it is necessary to calculate:

$$X = \frac{x \cdot Z}{f} \tag{4.3}$$

$$Y = \frac{y \cdot Z}{f} \tag{4.4}$$

The value of the focal length was obtained from the robot's documentation [85]. This is the position of the target object with respect to the camera reference frame.

To obtain the orientation of an object in 3D space starting from a 2D image, it is necessary to have additional information because the orientation is related to the rotation of the object with respect to the camera. To simplify the problem, in the orientation of the bottle we only considered that it is always in a vertical position.

At this point, we can formalize the information gathered so far as follows:

- The position and orientation of the camera with respect to the robot's reference frame can be associated with a homogeneous transformation from the camera to the robot's base, denoted as $T_{C\to B}$
- The position and orientation of the object (bottle) with respect to the camera's reference frame can be associated with a homogeneous transformation from the object to the camera, denoted as $T_{O\to C}$

What we aim to obtain, therefore, is the homogeneous transformation from the object to the robot's base $T_{O\to B}$. Each homogeneous transformation T is a 4×4 matrix of the form:

$$T = \begin{bmatrix} R & t \\ 0\ 0\ 0 & 1 \end{bmatrix} \tag{4.5}$$

where:

- $R \in \mathbb{R}^{3\times3}$ is the rotation matrix.
- $t \in \mathbb{R}^3$ is the translation vector.

We can rewrite the known data in matrix form:

- Position and orientation of the camera relative to the base:

$$T_{C\to B} = \begin{bmatrix} R_{C\to B} & t_{C\to B} \\ 0 & 1 \end{bmatrix} \tag{4.6}$$

- Position and orientation of the object relative to the camera:

$$T_{O\to C} = \begin{bmatrix} R_{O\to C} & t_{O\to C} \\ 0 & 1 \end{bmatrix} \tag{4.7}$$

The matrices should be multiplied:

$$T_{O\to B} = \begin{bmatrix} R_{C\to B} & t_{C\to B} \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} R_{O\to C} & t_{O\to C} \\ 0 & 1 \end{bmatrix} \tag{4.8}$$

$$T_{O\to B} = \begin{bmatrix} R_{C\to B}R_{O\to C} & R_{C\to B}t_{O\to C} + t_{C\to B} \\ 0 & 1 \end{bmatrix} \tag{4.9}$$

Therefore, we finally obtain:

- Rotation of the object relative to the base:

$$R_{O\to B} = R_{C\to B}R_{O\to C} \tag{4.10}$$

- Position of the object relative to the base:

$$t_{O\to B} = R_{C\to B}t_{O\to C} + t_{C\to B} \tag{4.11}$$

4.7 Path planning module

As shown in Fig. 4.1, the proposed grasping system is divided into two phases: one in which the approach to the target object and the estimation of the best possible position for grasping are performed in a more approximate and coarse manner, and a second phase, in which this process is repeated but based on images captured by the bottom camera, which provides a more detailed view but can only be used when the target is within a limited distance from the robot.

Algorithm 1 Path Planning algorithm for grasping

```
1: Input: id_camera_top, id_camera_bottom, proximity_threshold, max_error
2: Output: None
3: procedure PathPlanning(id_camera_top, id_camera_bottom)
4:     distance ← PathPlanning1(id_camera_top)
5:     PathPlanning2(distance, proximity_threshold, max_error,
6:         id_camera_bottom)
7: end procedure
```

Algorithm 1, summarizes this concept by presenting the main function, `PathPlanning`, which coordinates the process by first calling `PathPlanning1`. This function calculates the distance between the robot and the object using the top camera (which is more approximate) and then calls `PathPlanning2`, which enables more precise movement planning using the bottom camera. This algorithm clearly leverages the functionalities already described in the sections concerning object detection, depth estimation, coordinate mapping and inverse kinematics.

Algorithm 2 Path Planning 1

```
1: Input: id_camera
2: Output: distance
3: procedure PathPlanning1(id_camera)
4:     robot_position ← GetCurrentPosition()
5:     object_position_top ← GetObjectPositionTopCamera(id_camera)
6:     distance ← calculate_distance(robot_position, object_position_top)
7:     MoveRobot(distance)
8:     robot_position ← GetCurrentPosition()
9:     object_position_top ← GetObjectPositionTopCamera(id_camera)
10:    return distance
11: end procedure
```

In the first part of Algorithm 2, the robot calculates its position and the position of the bottle after taking images with the top camera. Then, the distance between these two points is calculated, considering that they are positioned within the same reference frame obtained after executing the coordinate mapping. The robot moves to cover this distance. At this point the robot position and the bottle's position are recalculated before initiating a more detailed movement planning phase.

Algorithm 3 Path Planning 2

```
1: Input: distance, proximity_threshold, max_error, id_camera
2: Output: None
3: procedure PathPlanning2(distance, proximity_threshold, max_error,
   id_camera)
4:     if distance ≤ proximity_threshold then
5:         repeat
6:             object_position_bottom ←
7:                 GetObjectPositionBottomCamera(id_camera)
8:             ideal_object_grasp_position ← GetIdealPosition()
9:             error ← CalculateError(robot_position, ideal_object_grasp_position)
10:            if error ≤ max_error then
11:                joint_values ←
12:                    CalculateInverseKinematics(object_position_bottom)
13:                ExecuteArmAndHandMovement(joint_values)
14:                break
15:            else
16:                robot_position ←
17:                    AdjustPosition(object_position_bottom, ideal_object_grasp_position)
18:            end if
19:        until error > max_error
20:    else
21:        robot_position ← GetCurrentPosition()
22:        object_position_bottom ←
23:            GetObjectPositionBottomCamera(id_camera)
24:        distance ← calculate_distance(robot_position, object_position_bottom)
25:        PathPlanning2(
26:            distance−proximity_threshold, proximity_threshold, max_error,
27:            id_camera)
28:    end if
29: end procedure
```

The precision phase then begins (Algorithm 3), relying on this last calculated distance, which serves as one of the inputs for this second phase, along with a proximity threshold that defines if the robot is close enough to grasp the object and a maximum error value (`max_error`) that indicates how much the robot's position can deviate from the ideal grasp position for the target object. If the distance between the robot and the bottle is less than the proximity threshold, the robot is close enough to initiate the grasping process. In this case, it enters an iterative position refinement loop. However, if the distance exceeds the threshold, the robot is too far and must recalculate the distance between itself and the object using the bottom camera, which provides a closer and more accurate view. The position of the object through the bottom camera can only be provided if the robot is close enough. In the position refinement loop, the robot evaluates how far its current object position is from the ideal position to grasp the object. If the resulting error is less than or equal to `max_error`, the robot is in the correct position and can proceed with the grasp. Otherwise, the robot adjusts its position and repeats the process until the error is small enough. When the error is acceptable, the inverse kinematics module is activated. This module receives the position of the target object as input and outputs the specific joint values required for Pepper's arm and hand to reach and grasp the object.

Basically, the robot approaches the bottle up to a certain distance. When it is close enough, it uses the bottom camera to get a better look at the bottle and calculates the ideal position to grab it. If the position is not perfect, it moves a little and recalculates. When everything is aligned, the robot moves its arm and graps the bottle.

4.8 Inverse kinematics module

As already mentioned in Section 2.2, inverse kinematics is defined as the problem of determining a set of appropriate joint configurations that allow the end effectors to move to desired positions as smoothly, rapidly, and accurately as possible. It represents the final step of our grasping system, enabling the execution of object grasping. Over the past decades, several methods and techniques have been proposed to provide fast and realistic solutions to the inverse kinematics (IK) problem, which is highly challenging because the equations describing it are often nonlinear, may admit multiple solutions, and are affected by singularities—specific configurations that limit the directions in which the robot and its arm can move [15].

In this work, the inverse kinematics module was implemented using Playful Kinematics [86], a framework designed to compute inverse kinematics in Python with a C++ backend. It was created to be user-friendly, prioritizing convenience and reactivity over absolute precision. It relies on Cyclic Coordinate Descent (CCD), an iterative heuristic search technique suitable for the interactive control of an articulated body, which will be described in detail in Section 4.8.1.

The framework can only be executed on Ubuntu and uses KDL (Kinematics and Dynamics Library) [87] to handle kinematics calculations. KDL is a library that is utilized in Playful Kinematics through ROS and provides an application-independent framework for modeling and computing kinematic chains, such as robots,

biomechanical human models, computer-animated figures, machine tools, etc. It offers class libraries for geometric objects (points, frames, lines, etc.), kinematic chains of various types (serial, humanoid, parallel, mobile, etc.), and their motion specification and interpolation.

The compilation of Playful Kinematics relies on Catkin [88], the ROS build system, and requires several other ROS packages to function, such as the urdf package [89], which contains a set of XML specifications for robot models, sensors, scenes, etc. Each XML specification has a corresponding parser in one or more languages.

To implement inverse kinematics for Pepper, Playful Kinematics takes into account not only the degrees of freedom that can be found in its arms (Fig. 4.15) —namely ShoulderRoll, ShoulderPitch, ElbowYaw, ElbowRoll and WristYaw —but also HipPitch, KneePitch and HipRoll , which are the degrees of freedom present in the lower part of its body (Fig. 4.16). These allow for a certain degree of mobility and flexibility, even though the robot does not have legs in the traditional sense and moves using wheels.

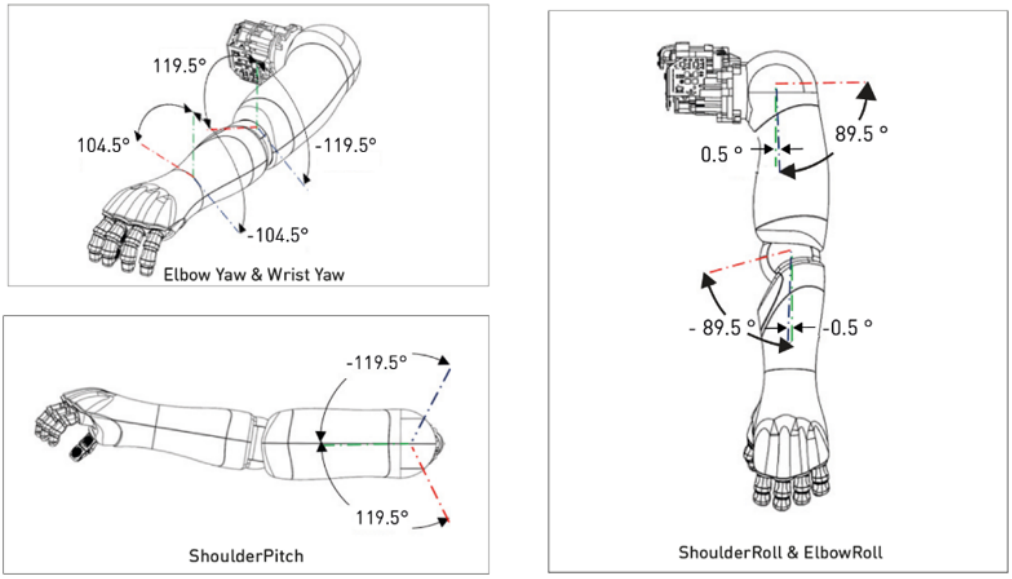

Figure 4.15 Degree of freedom in Pepper's arms [58].

The reason for this choice is tied to the fact that, since Pepper's arm is equipped with only 5 degrees of freedom, there are few points the robot can reach by moving its arms in space. Essentially, it has a limited workspace, and the degree of freedom in the lower body allow this workspace to be expanded. Additionally, incorporating the degrees of freedom in the lower body allows the inverse kinematics algorithm to identify more efficient or natural solutions for arm movement. For instance, rather than contorting the arm into an awkward position, the algorithm can subtly adjust the torso's tilt to reach the target with reduced effort or smoother motion.

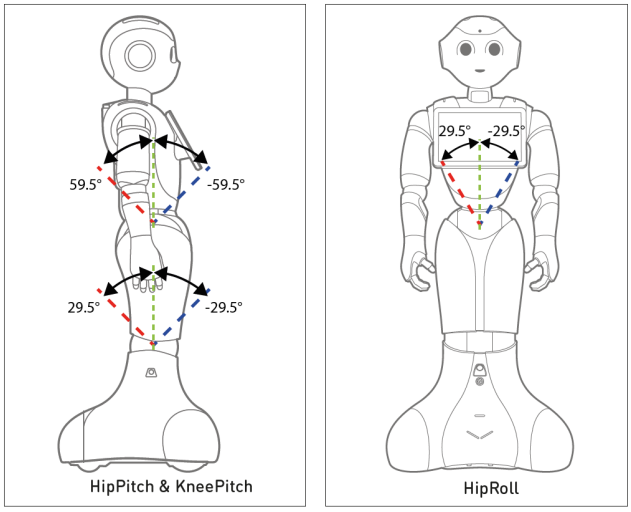

Figure 4.16 Degree of freedom in Pepper's "legs". The robot's base is not considered. [58].

As mentioned, it is not possible to control each individual finger of Pepper's hand; instead, we can only decide whether to open or close the hand. For this reason, inverse kinematics does not take the hand into account.

Specifically, Playful Kinematics requires the initial position in radians of the aforementioned 8 degrees of freedom and the target position to be reached (i.e., the position of the bottle) to calculate the inverse kinematics. It is also necessary to specify the desired orientation of the end-effector. If no solution is found, the returned posture is still meaningful, representing the robot's best effort to approach the target.

4.8.1 Cyclic Coordinate Descent (CCD)

Cyclic Coordinate Descent (CCD) is a widely used iterative algorithm for solving inverse kinematics (IK) problems. It is implemented in various computer graphics and robotics applications and is particularly prevalent in the computer games industry. Beyond gaming, CCD has also been effectively used in protein science for protein structure prediction and/or structure determination [90]. The algorithm is straightforward to implement, computationally fast, and numerically stable, with a linear-time complexity proportional to the number of degrees of freedom (DoF). CCD minimizes position and orientation errors by adjusting one joint variable at a time. Starting from the end effector and moving inward toward the manipulator base, the algorithm transforms each joint sequentially to bring the end effector as close as possible to the target. This process repeats until a satisfactory solution is achieved. The low computational cost per joint enables rapid solution formulation. Fig. 4.17 illustrates the IK problem-solving process using the CCD algorithm across multiple iterations.

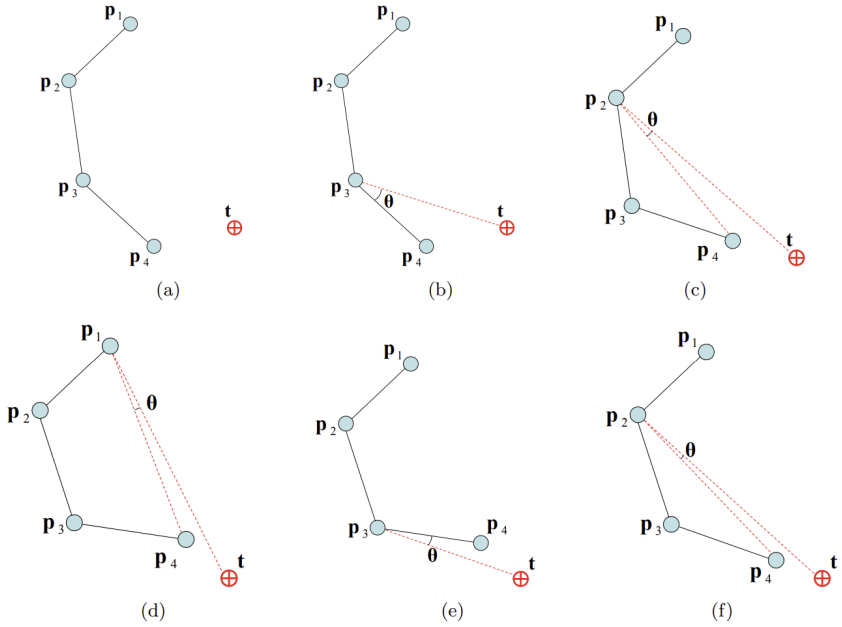

Figure 4.17 An example of visual solution of the IK problem using the CCD algorithm. [16]

In step (a), the initial position of the manipulator and the target are shown. In the first iteration (b), the algorithm calculates the angle θ between the end effector, joint p_3, and the target, then rotates joint p_4 by this angle. In the second iteration (c), the angle θ is computed between the end effector, joint p_2, and the target, prompting the rotation of joints p_4 and p_3 by this angle. Steps (d), (e), and (f) repeat this process for as many iterations as necessary. The algorithm terminates when the end effector reaches the target or comes sufficiently close.

Like other inverse kinematics solvers, CCD can produce multiple possible postures from a single initial configuration, making it challenging to select a feasible posture from the resulting set. To address this, manipulator constraints must be applied to limit motion to practical configurations. Despite its speed, CCD has notable drawbacks. It often generates motion with erratic discontinuities and tends to overemphasize the movement of joints closer to the end effector in the kinematic chain. This can result in unnatural motion, even when constraints are applied.

Chapter 5

Experimental results

In this chapter, the experimental results of the study are presented. Section 5.1 evaluates the effectiveness of the object detection module, implemented using YOLOv10, in detecting the target object. Section 5.2 compares the performance of Depth Anything v1 and Depth Anything v2 in estimating the distance between the target object and the robot. Additionally, it will be explained how the values of the `max_depth` parameter for Depth Anything v2 were determined experimentally. Finally, Section 5.3 examines the performance of the grasping system, measured by the success rate of grasping the target object over multiple attempts, and explores how these experiments were conducted in QiBullet.

The experiments were all run on a laptop with an Intel Core i9 processor and an Nvidia RTX 4060 GPU. The vision modules (object detection and depth estimation) are executed on a Flask server running on Windows, while QiBullet and ROS were installed on Ubuntu 18.04.6 LTS, running via Windows Subsystem for Linux (WSL).

5.1 Object detection module evaluation

To evaluate the object detection module, a dataset was created by collecting synthetic images from the robot's cameras (top and bottom) using the QiBullet simulator (Fig. 5.1). The model used to implement this module, YOLOv10, like all other versions of YOLO, is not natively trained on synthetic data; therefore, assessing its performance in this scenario is particularly interesting.

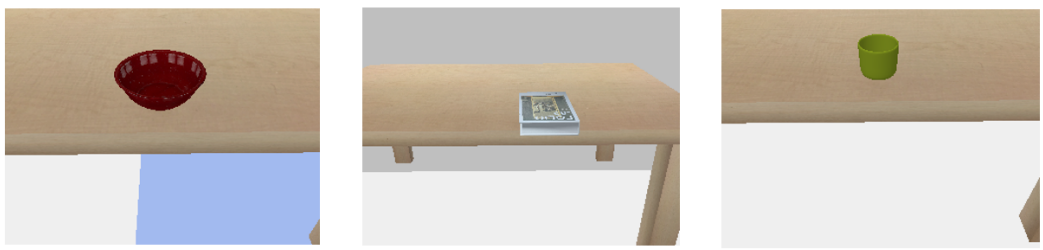

Figure 5.1 Sythetic images captured to create the dataset.

The dataset consists of 300 images: 150 depicting bottles and 150 depicting other objects that YOLO can detect without additional training, as they are already present in the COCO dataset [91], on which YOLO is pre-trained. The objects chosen to form this second class of the dataset, which we will call "not bottle", are an apple, a cup, a tennis ball, a bowl and a book. All these models were first loaded into the simulator using URDF files and the images were then captured from various angles and distances using both the bottom and the top camera.

The objective of this test is to verify whether the detection of the chosen target object is effectively robust, taking into account different positions from which the photo is taken and assessing whether it might be confused with other objects that could have a similar geometric shape or structure. Assuming that images containing at least one bottle are classified as the positive class, and images of non-bottles are classified as the negative class, we can consider the following values:

- **True negatives (TN)**: number of images without bottles in which the model correctly indicated the absence of a bottle.
- **True positives (TP)**: number of images with bottles in which the model correctly detected a bottle.
- **False positives (FP)**: number of images without bottles in which the model erroneously detected a bottle.
- **False negatives (FN)**: number of images with a bottle present in which the model failed to detect any bottle.

Through these values, 3 very important metrics have been calculated:

- **Precision**: measures the percentage of correct positive predictions out of all positive predictions made by the model. It is calculated as

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.1}$$

- **Recall**: measures the percentage of true positives detected by the model out of all actual true positives. It is calculated as

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.2}$$

- **Accuracy**: measures the overall percentage of correct predictions (both positive and negative) out of the total predictions. It is calculated as

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.3}$$

In Table 5.1 are shown the results of applying YOLOv10 to the 150 images of the positive class "bottle" and to the 150 images of the negative class "not bottle".

Metric	Value
Precision	99.33%
Recall	100%
Accuracy	99.66%

Table 5.1 Performance metrics of the model.

In general, by observing the confusion matrix in Fig. 5.2, we can observe that the model has almost never made a prediction error, correctly identifying all the true negative examples and nearly all the true positive ones. Only in one case a bottle was detected where none was actually present, while there were never any instances in which bottles were not correctly detected.

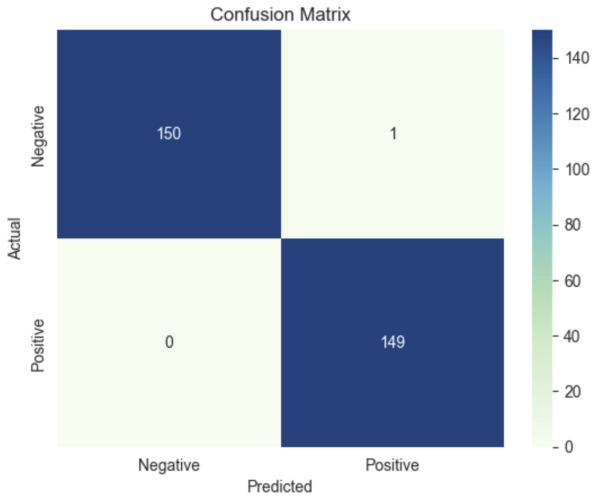

Figure 5.2 Confusion matrix.

These metrics confirm the high reliability of YOLO, even on synthetic data, such as those obtained from a simulator.

5.2 Depth estimation module evaluation

In Section 4.5 and 4.5.1, the monocular depth estimation models Depth Anything V1 and Depth Anything V2 were introduced. Initially, in the proposed system, the depth estimation module was implemented using Depth Anything V1. Following the release of the new model, a comparison between the two models was conducted before changing the implementation, to determine which was more suitable for the system. Specifically, it was crucial to understand which of the two was more efficient in metric depth estimation. The main difference between Depth Anything V1 and V2 lies in the introduction, in version 2, of the `max_depth` parameter, which allows the model to be adapted to the desired metric scale and specific environments (indoor or outdoor). To compare the two models, we will first analyze the performance of Depth Anything V1, then examine how the effectiveness of Depth Anything V2 varies with changes in the `max_depth` parameter. These tests will be conducted by evaluating performance on both the top camera and the bottom camera. As previously noted, the top camera provides a wider field of view but cannot detect nearby objects, even when the robot's head is tilted to its maximum extent, whereas the bottom camera has a more limited field of view but offers a more realistic perspective of the observed objects. Finally, the best `max_depth` values for the top camera and bottom camera will be identified, and these results will be compared with those obtained from Depth Anything V1.

The metric used to evaluate the performance of Depth Anything V1 and V2 for metric depth estimation is RMSE (Root Mean Square Error). This is the same metric used by the authors of these models in their respective papers [79] and [80] to compare the performance of their models with others in the literature. This metric measures the square root of the average of the squared differences between the predicted values and the ground truth:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{5.4}$$

In depth estimation, RMSE is particularly used because it helps identify models that make significant errors. By squaring the errors, larger errors are penalized more heavily. The RMSE is a scalar metric (a single number), making it ideal to compare different models or approaches. A model with a lower RMSE is generally considered better. For each model, the RMSE value will be analyzed at different distances. The distance under consideration is simply the distance between the target object and the robot. For the bottom camera, the target distance values considered are relatively close to each other, because it has a fairly limited field of view, resulting in few positions where it can be used effectively. According to the experiments conducted in this work, to see the object with the bottom camera, it is necessary to be at a maximum distance of 70 cm from it.

In these experiments to calculate the RMSE value, 30 observations were considered each time. Clearly, not all possible values of `max_depth` were taken into account and testing was limited to `max_depth=20`, as this is the value recommended by the authors of Depth Anything v2 for indoor environments. Additionally, some values were skipped because, in many cases, increasing the `max_depth` value by 1

does not lead to significant changes in the results.

Target distance	RMSE
50 cm	0.293
40 cm	0.294
30 cm	0.311

Table 5.2 Results of Depth Anything v1 with images captured from bottom camera for various target distances.

The RMSE values shown in Table 5.2 for the bottom camera are mostly good but not exceptional. As for the top camera, it was decided to consider greater distances between points, because, since it is a camera with a wider field of view, detecting significant differences in the evaluation results requires larger movements and covering more distance.

Target distance	RMSE
2 m	0.716
1.5 m	0.019
1 m	0.047

Table 5.3 Results of Depth Anything v1 with images captured from top camera for various target distances.

The values for the distances of 1 m and 1.5 m shown in Table 5.3 are really excellent, but it can be noted that there is a certain tendency to obtain worse results as the distance increases.

The same distance values were also considered for the tests on Depth Anything v2. However, in this case, the variation of RMSE with respect to the `max_depth` parameter was analyzed. The results obtained for the bottom camera are shown in Table 5.4.

Target Distance	max_depth								
	5	7	9	11	13	15	17	19	20
50 cm	0.161	0.054	0.017	0.163	0.272	0.372	0.486	0.596	0.656
40 cm	0.133	0.051	0.030	0.116	0.196	0.280	0.363	0.443	0.477
30 cm	0.093	0.037	0.017	0.070	0.126	0.185	0.238	0.298	0.327

Table 5.4 Results of Depth Anything v2 with images captured from bottom camera expressed in RMSE for various target distances and values of the `max_depth` parameter.

It can be observed, especially from Fig. 5.3, that the best value for `depth_map` for the bottom camera is `max_depth`=9, as it exhibits the best performance across varying distances. It is also notable that for the bottom camera, the RMSE values

tend to decrease up to `max_depth`=9 and then increase rapidly, leading to an increasingly inaccurate model. This issue is likely due to the bottom camera perceiving the environment and objects at very close range. As a result, setting the maximum estimation limit to a value like 20 meters is excessive and detrimental to the model's performance.

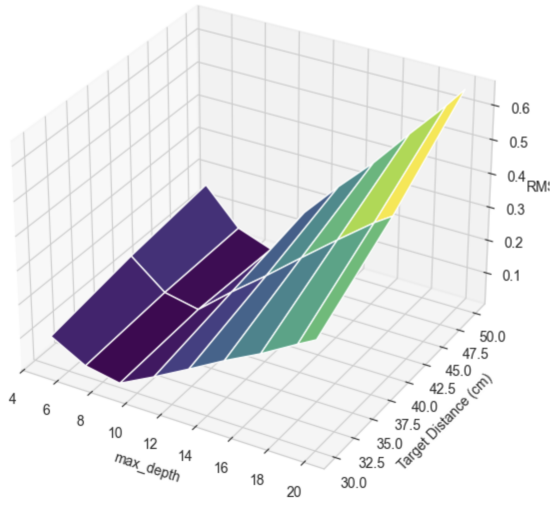

Figure 5.3 Variation of RMSE based on `max_depth` and target distance (bottom camera)

As for the top camera (Table 5.5), it is not easy to determine which `max_depth` value is the best. `max_depth`=19 is the optimal value for medium distances, such as 1 m or 1.5 m, while `max_depth`=20 performs better over longer distances. The choice of the best parameter in this case also depends heavily on the context in which it is being used: if the robot needs to cover large distances, the best value is `max_depth`=20; otherwise, it is better to consider `max_depth`=19.

Target Distance	**max_depth**								
	5	7	9	11	13	15	17	19	20
2 m	1.398	1.206	1.012	0.821	0.628	0.440	0.236	0.103	0.047
1.5 m	1.018	0.873	0.728	0.582	0.437	0.297	0.152	0.010	0.077
1 m	0.624	0.522	0.415	0.310	0.210	0.117	0.008	0.103	0.150

Table 5.5 Results of Depth Anything v2 with images captured from top camera expressed in RMSE for various target distances and values of the `max_depth` parameter.

Focusing on a more detailed analysis and also observing the graph in Fig. 5.4, it is possible to notice how smaller `max_depth` values clearly exhibit the worst performance. This is likely due to the fact that they impose a limit at an excessively short distance, preventing a proper evaluation of the depth value. As the `max_depth` value increases, performance improves—not only for average distances but especially when considering a target distance of 2 meters.

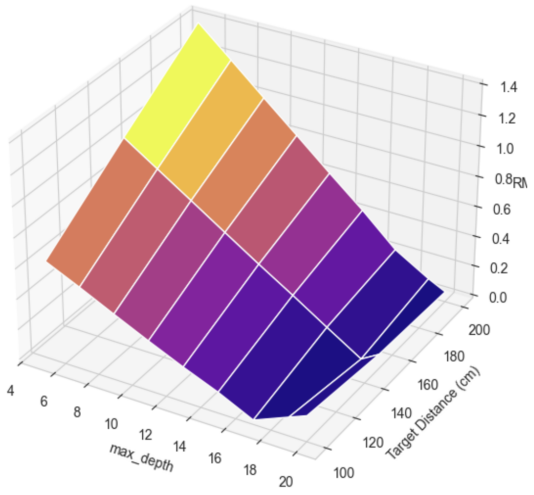

Figure 5.4 Variation of RMSE based on `max_depth` and target distance (top camera)

To compare Depth Anything v1 and v2, the best `max_depth` values for version 2 were used, namely `max_depth`=9 for the bottom camera and `max_depth`=19 for the top camera. By also examining the graphs in Fig. 5.5, it can be observed that Depth Anything v2 consistently outperforms its version 1 across all examined distances.

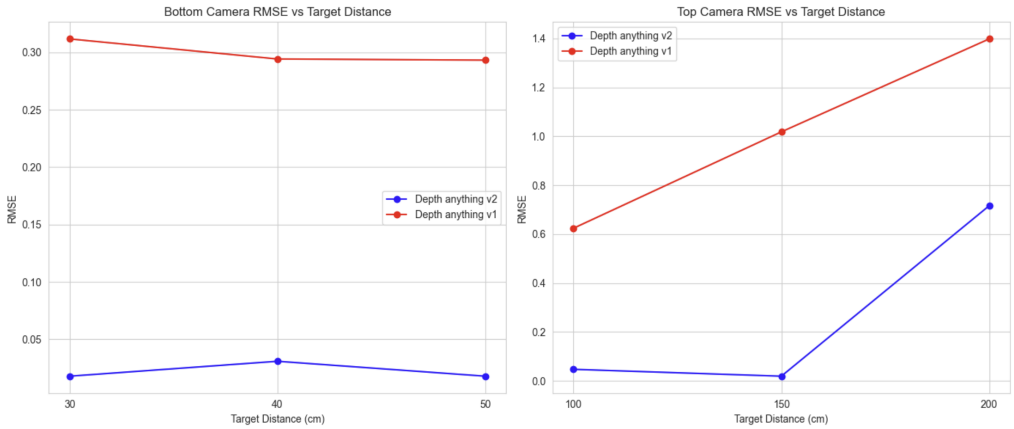

Figure 5.5 Comparison of the performance of Depth Anything v1 and Depth Anything v2.

In light of the results of these experiments, Depth Anything v2 has become the cornerstone of the depth estimation module.

5.3 Results of grasping attempts with Pepper

To test the effectiveness of the grasping system as a whole, a series of simulations were conducted using the QiBullet simulator, which has already been extensively described throughout this thesis. The simulated environment in which the grasping was tested resembles an office, as shown in Fig. 5.6.

A demonstration video can be found at:
`https://drive.google.com/file/d/1-bnVR9A43_qvN2H7tZ9QyVUDxzcZhbbf/view`

A grasp is considered successful when the simulated robot is able to reach and lift the bottle. Conversely, a grasp is deemed failed if the simulated robot cannot grasp and lift the bottle.

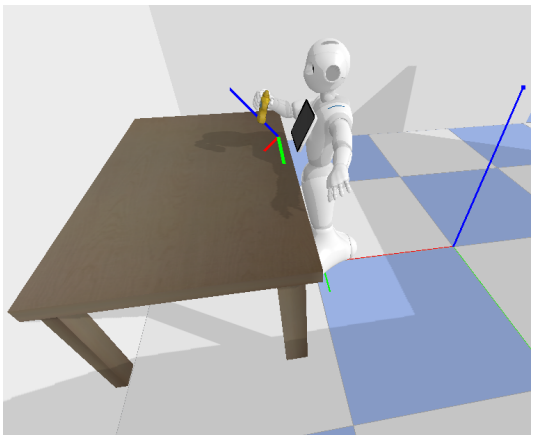

Figure 5.6 A successful object grasp in QiBullet simulator.

The simulated tests were carried out following these steps:

1. Generate a random reachable position for the bottle, which must always remain on the simulated table, as the robot is unable to bend forward effectively or reach objects placed too high.
2. Initiate the path planning algorithm to bring Pepper closer to the object.
3. If the robot is sufficiently close to the bottle, generate possible kinematic solutions that would allow Pepper to successfully reach and grasp the bottle.
4. Otherwise, generate a new position/orientation for the bottle and repeat the process.

As expected, the positions yielding a positive outcome were those where the bottle was spawned closer to the table's edges, where Pepper's arm can more easily reach the target object. The time performance proved suitable for real-time applications, requiring approximately 40 seconds to complete the reaching and grasping tasks. However, this value can vary depending on whether the robot is closer or farther from the target object. A total of 25 simulations were performed, with the robot starting from different positions, yielding results shown in Table 5.6.

Result	Count
Failed grasp	7
Successful grasp	18

Table 5.6 Results of grasping attempts with Pepper

This translates to a success rate—i.e., the percentage of successful grasps relative to the total number of simulations—of 72%. However, it should be noted that in the simulator lighting variations, which are critical in a computer vision-based system, differ significantly from those in real-world conditions, and the synthetic images captured lack noise or other distortions that may occur in reality. Additionally, the robot's movements are always extremely precise, and there is no friction—for example, with certain types of flooring—that could limit Pepper's precise movements in real-world scenarios. Nevertheless, the obtained result is highly satisfactory, especially considering that no markers or 3D models were used in the proposed approach and that technologies like Depth Anything, which have not previously been employed in the literature for this type of task, were utilized.

Chapter 6

Conclusions

This work introduces a new application for the Pepper robot: grasping unknown objects using object detection and state-of-the-art metric depth estimation techniques. To date, there are no reported results of other approaches implemented on Pepper for grasping objects without relying on markers, 3D models or additional depth cameras. The integration of advanced computer vision techniques with classic robotic algorithms enables the final system to be both robust and efficient. Thanks to the modularity of the implementation, it can be easily adapted to other humanoid robots. Despite the platform limitations, the grasping system proves to be reliable and suitable for real-time tasks.

This work opens up many interesting future developments. The first step is to enable the robot to identify and grasp objects of various shapes and sizes, rather than just bottles. Currently, the system allows the robot to perform only single-handed grasps. However, given the weight limitations of what Pepper can lift with one arm, developing an option that allows the system to use one or both arms depending on the task appears to be a highly promising challenge. Additionally, it would be interesting to enhance the obstacle detection and collision detection in the system and to introduce the possibility of using voice commands to instruct the robot to retrieve an object for you.

Acknowledgments

First and foremost, I would like to express my gratitude to Professor Luigi Cinque, who gave me the opportunity to experience this wonderful journey.

Secondly, a heartfelt thank you to all the members of Sapienza VisionLab, especially Marco, with his many advices and constant encouragement to never give up, and Omar, who was always available to give me little robotics lessons even when he was busy with work.

Thirdly, my gratitude goes to my boyfriend, my family and friends, who supported me through the years and helped especially during the most difficult times.

Bibliography

[1] Michael Han and Cindy Harnett. Journey from human hands to robot hands: biological inspiration of anthropomorphic robotic manipulators. *Bioinspiration Biomimetics*, 2024.

[2] Minzhou Luo. Finger orientation for robotic hands. *Springer London*, 2013.

[3] T. Asfour, K. Regenstein, P. Azad, J. Schroder, A. Bierbaum, N. Vahrenkamp, and R. Dillmann. Armar-iii: An integrated humanoid platform for sensory-motor control. *2006 6th IEEE-RAS International Conference on Humanoid Robots*, 2006.

[4] A. Konno, K. Nagashima, R. Furukawa, K. Nishiwaki, T. Noda, M. Inaba, and H. Inoue. Development of a humanoid robot saika. *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS '97*, 1997.

[5] Uikyum Kim, Dawoon Jung, Heeyoen Jeong, Jongwoo Park, Hyun-Mok Jung, Joono Cheong, Hyouk Choi, Hyunmin Do, and Chanhun Park. Integrated linkage-driven dexterous anthropomorphic robotic hand. *Nature Communications*, 2021.

[6] Zhe Xu and Emanuel Todorov. Design of a highly biomimetic anthropomorphic robotic hand towards artificial limb regeneration. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016.

[7] Steffen Puhlmann, Jason Harris, and Oliver Brock. Rbo hand 3 – a platform for soft dexterous manipulation. *IEEE Transactions on Robotics*, 2022.

[8] Filomena Simone, Gianluca Rizzello, Stefan Seelecke, and Paul Motzki. A soft five-fingered hand actuated by shape memory alloy wires: Design, manufacturing, and evaluation. *Frontiers in Robotics and AI*, 2020.

[9] Cecilia Laschi, Paolo Dario, Maria Chiara Carrozza, Eugenio Guglielmelli, Giancarlo Teti, Davide Taddeucci, Fabio Leoni, Bruno Massa, Massimiliano Zecca, and Roberto Lazzarini. Grasping and manipulation in humanoid robotics. *IEEE*, 2000.

[10] VisionLab, Research laboratory of the Department of Computer Science at the Sapienza University of Rome, Italy.

[11] Sulabh Kumra and Christopher Kanan. Robotic grasp detection using deep convolutional neural networks. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[12] Minglun Dong and Jian Zhang. A review of robotic grasp detection technology. *Robotica*, 2023.

[13] John J. Craig. *Introduction to Robotics: Mechanics and Control*. Pearson Prentice Hall, 3rd edition, 2005.

[14] Bruno Siciliano, L. Sciavicco, Villani Luigi, and G. Oriolo. Robotics: Modelling, planning and control. *Advanced Textbooks in Control and Signal Processing*, 2009.

[15] M.W. Spong. Robot modeling and control. *Industrial Robot: An International Journal*, 2006.

[16] Andreas Aristidou and Joan Lasenby. Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver. *University of Cambridge, Department of Engineering*, 2009.

[17] Chaima Lahdiri, Houssem Saafi, Abdelfattah Mlika, and Med Amine Laribi. Forward kinematic model resolution of a new spherical parallel manipulator. *Robotics and Mechatronics*, 2024.

[18] Mahmoud Gouasmi. Robot kinematics, using dual quaternions. *IAES International Journal of Robotics and Automation (IJRA)*, 2012.

[19] Bharath Reddy Lakki Reddy Venkata and Himanth M. Forward kinematics analysis of robot manipulator using different screw operators. *International Journal of Robotics and Automation*, 2018.

[20] A. Balestrino, G. De Maria, and L. Sciavicco. Robust control of robotic manipulators. *IFAC Proceedings Volumes*, 1984.

[21] Charles W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 1986.

[22] Yoshihiko Nakamura and Hideo Hanafusa. Inverse kinematic solutions with singularity robustness for robot manipulator control. *Journal of Dynamic Systems, Measurement, and Control*, 1986.

[23] Samuel Buss and Jin-Su Kim. Selectively damped least squares for inverse kinematics. *J. Graphics Tools*, 2005.

[24] R. Fletcher. Newton-like methods. *Practical Methods of Optimization*, 2000.

[25] C. Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. *Canadian theses on microfiche*, 1993.

[26] Dan Ding, Yun-Hui Liu, and Shuguo Wang. Computing 3-d optimal form-closure grasps. *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, 2000.

[27] V.-D. Nguyen. Constructing force-closure grasps. *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, 1986.

[28] A.T. Miller, S. Knoop, H.I. Christensen, and P.K. Allen. Automatic grasp planning using shape primitives. *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, 2003.

[29] Claire Dune, Eric Marchand, Christophe Collowet, and Christophe Leroux. Active rough shape estimation of unknown objects. *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008.

[30] Renaud Detry, Carl Henrik Ek, Marianna Madry, Justus Piater, and Danica Kragic. Generalizing grasps across partly similar objects. *2012 IEEE International Conference on Robotics and Automation*, 2012.

[31] Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps. *International Journal of Robotics Research*, 2013.

[32] Joseph Redmon and Anelia Angelova. Real-time grasp detection using convolutional neural networks. *Proceedings - IEEE International Conference on Robotics and Automation*, 2014.

[33] Shehan Caldera, Alexander Rassau, and Douglas Chai. Review of deep learning methods in robotic grasp detection. *Multimodal Technologies and Interaction*, 2018.

[34] Joe Watson, Josie Hughes, and Fumiya Iida. Real-world, real-time robotic grasping with convolutional neural networks. *Towards Autonomous Robotic Systems*, 2017.

[35] Antonio Alliegro, Martin Rudorfer, Fabio Frattin, Aleš Leonardis, and Tatiana Tommasi. End-to-end learning to grasp via sampling from object point clouds. *IEEE Robotics and Automation Letters*, 2022.

[36] Markus Przybylski, Nikolaus Vahrenkamp, Tamim Asfour, and Rüdiger Dillmann. Grasp and motion planning for humanoid robots. *Grasping in Robotics*, 2013.

[37] Matei Ciocarlie, Corey Goldfeder, and Peter Allen. Dimensionality reduction for hand-independent dexterous robotic grasping. *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007.

[38] Corey Goldfeder, Matei Ciocarlie, Jaime Peretzman, Hao Dang, and Peter K. Allen. Data-driven grasping with partial sensor data. *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009.

[39] Corey Goldfeder, Matei Ciocarlie, Hao Dang, and Peter K. Allen. The columbia grasp database. *2009 IEEE International Conference on Robotics and Automation*, 2009.

[40] James Kuffner and Steven LaValle. Rrt-connect: An efficient approach to single-query path planning. *Proceedings - IEEE International Conference on Robotics and Automation*, 2000.

[41] Evan Drumwright and Victor Ng-Thow-Hing. Toward interactive reaching in static environments for humanoid robots. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.

[42] Michael Gienger, Marc Toussaint, Nikolay Jetchev, Achim Bendig, and Christian Goerick. Optimization of fluent approach and grasp motions. *Humanoids 2008 - 8th IEEE-RAS International Conference on Humanoid Robots*, 2008.

[43] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 1986.

[44] Bogdan Gabriel Draghici, Alexandra Elena Dobre, Marius Misaros, and Ovidiu Petru Stan. Development of a human service robot application using pepper robot as a museum guide. *2022 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2022.

[45] Misao Miyagawa, Yuko Yasuhara, Tetsuya Tanioka, Rozzano Locsin, Waraporn Kongsuwan, Elmer Catangui, and Kazuyuki Matsumoto. The optimization of humanoid robot's dialog in improving communication between humanoid robot and older adults. *Intelligent Control and Automation*, 2019.

[46] Gavin Suddrey, Adam Jacobson, and Belinda Ward. Enabling a pepper robot to provide automated and interactive tours of a robotics laboratory. *ArXiv*, 2018.

[47] Z. Al barakeh, S. alkork, A. S. Karar, S. Said, and T. Beyrouthy. Pepper humanoid robot as a service robot: a customer approach. *2019 3rd International Conference on Bio-engineering for Smart Technologies (BioSMART)*, 2019.

[48] Hansol Woo, Gerald K. LeTendre, Trang Pham-Shouse, and Yuhan Xiong. The use of social robots in classrooms: A review of field-based studies. *Educational Research Review*, 2021.

[49] Fumihide Tanaka, Kyosuke Isshiki, Fumiki Takahashi, Manabu Uekusa, Rumiko Sei, and Kaname Hayashi. Pepper learns together with children: Development of an educational application. *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, 2015.

[50] Florian Lier and Sven Wachsmuth. Towards an open simulation environment for the pepper robot. *Companion of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*, 2018.

[51] João Silva, Miguel Simão, Nuno Mendes, and Pedro Neto. Navigation and obstacle avoidance: a case study using pepper robot. *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, 2019.

[52] Naomarks, Aldebaran Robotics, `http://doc.aldebaran.com/2-5/naoqi/vision/allandmarkdetection.html#allandmarkdetection`.

[53] Paola Ardón, Mauro Dragone, and Mustafa Suphi Erden. Reaching and grasping of objects by humanoid robots through visual servoing. *EuroHaptics*, 2018.

[54] Hongbin Chen. Target positioning and grasping of nao robot based on monocular stereo vision. *Mobile Information Systems*, 2022.

[55] Yingrui Jin, Zhaoyuan Shi, Xinlong Xu, Guang Wu, Hengyi Li, and Shengjun Wen. Target localization and grasping of nao robot based on yolov8 network and monocular ranging. *Electronics*, 2023.

[56] Judith Müller, Udo Frese, Thomas Röfer, Rodolphe Gelin, and Alexandre Mazel. Graspy - object manipulation with nao. *Gearing Up and Accelerating Cross-fertilization between Academic and Industrial Robotics Research in Europe*, 2014.

[57] Giovanni Claudio, Fabien Spindler, and François Chaumette. Vision-based manipulation with the humanoid robot romeo. *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, 2016.

[58] Pepper technical specifications, Aldebaran Robotics, `https://support.unitedrobotics.group/en/support/solutions/articles/80000958735-pepper-technical-specifications`.

[59] Zuria Bauer, Felix Escalona, Edmanuel Cruz, Miguel Cazorla, and Francisco Gomez-Donoso. Refining the fusion of pepper robot and estimated depth maps method for improved 3d perception. *IEEE Access*, 2019.

[60] Comparison of Pepper's OS versions FAQ, Aldebaran Robotics, `https://drive.google.com/file/d/13JDe-LtH_Li7dNHRYeMLd5elBQI1E1gC/view?usp=drive_link`.

[61] Comparison of Pepper's OS versions datasheet, Aldebaran Robotics, `https://drive.google.com/file/d/1gfD6moNim2uNtO3JZmEJKVhnvPtKA4s3/view?usp=drive_link`.

[62] Pepper Python SDK documentation, Aldebaran Robotics, `http://doc.aldebaran.com/2-5/dev/python/index.html`.

[63] Pepper C++ SDK documentation, Aldebaran Robotics, `http://doc.aldebaran.com/2-5/dev/cpp/index.html`.

[64] Pepper QiSDK documentation, Softbank Robotics, `https://qisdk.softbankrobotics.com/sdk/doc/pepper-sdk/index.html`.

[65] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, 2004.

[66] Eric Rohmer, Surya P. N. Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013.

[67] Olivier Michel. Cyberbotics ltd. webots™: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 2004.

[68] E. Pot, J. Monceaux, R. Gelin, and B. Maisonnier. Choregraphe: a graphical tool for humanoid robot programming. *RO-MAN 2009 - The 18th IEEE International Symposium on Robot and Human Interactive Communication*, 2009.

[69] Erwin Coumans, Bullet physics engine. Open Source Software: `http://bulletphysics.org`.

[70] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`.

[71] QiBullet API documentation, Maxime Busy and Maxime Caniot, Softbank Robotics Research, `https://softbankrobotics-research.github.io/qibullet/annotated.html`.

[72] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 2009.

[73] Berk Calli, Arjun Singh, James Bruce, Aaron Walsman, Kurt Konolige, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. Yale-cmu-berkeley dataset for robotic manipulation research. *The International Journal of Robotics Research*, 2017.

[74] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[75] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[76] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[77] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 2012.

[78] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 1995.

[79] Lihe Yang, Bingyi Kang, Zilong Huang, Xiaogang Xu, Jiashi Feng, and Hengshuang Zhao. Depth anything: Unleashing the power of large-scale unlabeled data. In *CVPR*, 2024.

[80] Lihe Yang, Bingyi Kang, Zilong Huang, Zhen Zhao, Xiaogang Xu, Jiashi Feng, and Hengshuang Zhao. Depth anything v2. *arXiv:2406.09414*, 2024.

[81] Yang, Lihe and Kang, Bingyi and Huang, Zilong and Xu, Xiaogang and Feng, Jiashi and Zhao, Hengshuang. Depth Anything V2 for Metric Depth Estimation, `https://github.com/DepthAnything/Depth-Anything-V2/tree/main/metric_depth`.

[82] Jinchang Ren, Amir Hussain, Jiangbin Zheng, Cheng-Lin Liu, and Bin Luo. Editorial: Special issue on recent advances in cognitive learning and data analysis. *Cognitive Computation*, 2021.

[83] Hao Fu and Rui Yu. Lidar scan matching in off-road environments. *Robotics*, 2020.

[84] Zhengqi Li and Noah Snavely. Megadepth: Learning single-view depth prediction from internet photos. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[85] 2D Cameras, Aldebaran Robotics, `http://doc.aldebaran.com/2-5/family/pepper_technical/video_2D_pep.html`.

[86] Playful Kinematics, Vincent Berenz, Github `https://github.com/vincentberenz/playful_kinematics`.

[87] Kinematics and Dynamics Library (KDL), Orocos project, `https://www.orocos.org/kdl.html`.

[88] Catikin, ROS official documentation, `http://wiki.ros.org/catkin/conceptual_overview`.

[89] urdf package, ROS official documentation, `http://wiki.ros.org/urdf`.

[90] Adrian Canutescu and Roland Dunbrack. Cyclic coordinate descent: A robotics algorithm for protein loop closure. *Protein science : a publication of the Protein Society*, 2003.

[91] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. *Computer Vision – ECCV 2014*, 2014.