TÉCNICO LISBOA

Domestic Robot Grasping using Visual-Servoing and Deep Learning

João Manuel de Almeida Correia Pereira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Pedro Manuel Urbano de Almeida Lima

Examination Committee

Chairperson: Prof. David Manuel Martins de Matos
Supervisor: Prof. Pedro Manuel Urbano de Almeida Lima
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

November 2020

Dedicated to my parents, for all their love and support.

Acknowledgments

I would like to thank my supervisor Prof. Pedro Lima for his help and guidance in this work, and for sparking my interest in robotics and inviting me to work at ISR. I also thank Dr. Meysam Basiri for his coordination of my research grant on robot benchmarking, where we defined and implemented the benchmark used on this thesis evaluation, among other interesting work.

I also give thanks to IDMind, especially Carlos Marques and Paulo Alvito for their help in brainstorming, designing and manufacturing parts to fit a new arm in the MBot robot.

I am grateful to Tiago Cardoso for his work in the arm's assembly, and for his technical support on the MBot platform. Additionally I thank the whole SocRob @Home team for their help and knowledge, and also for the company and joyful moments.

Finally I extend my gratitude to my father, for always looking after me, and without whose support I never could have done this work, and my mother for always loving me.

Resumo

O manuseamento robótico é uma funcionalidade importante em robôs domésticos. Esta área tem sido objecto de extensa investigação levando ao desenvolvimento de diferentes métodos, desde sistemas baseados na teoria de controlo, até sistemas que usam somente aprendizagem automática.

Esta tese propõe um processo para manipulação de objectos, cujo componente central usa seguimento visual (visual-servoing ou VS) para posicionar o efector terminal do braço robótico junto ao objecto a agarrar, com localização obtida através de retroação visual. Uma câmara com sensor de profundidade, fixa à cabeça de um robô móvel, observa o objecto e o efector terminal e estima ambas as posições, usadas para calcular o erro posicional de forma independente da calibração cinemática e da câmara. O erro é minimizado através de controlo proporcional do braço, que é actuado através de cinemática diferencial.

A posição do objecto é obtida usando uma rede neuronal convolucional que identifica a área da imagem que contém o objecto. A profundidade no centro desta área é usada para calcular a posição. O efector terminal é localizado usando marcadores de realidade aumentada (AR) colocados à volta do pulso.

O processo completo obtém o volume de colisão do espaço, move o braço até uma posição de pré-agarramento que faz com que o efector terminal fique visível na câmara, e o seguimento visual é activado.

O sistema demonstra sucesso em agarrar objectos domésticos em diferentes posições, obtendo bons resultados num teste de manuseamento para usar em futuras competições da European Robotics League.

Palavras-chave: Robótica, Manipulação, Visual-Servoing

Abstract

Object grasping and manipulation are important capabilities for domestic service robots. Extensive research work has been done in this area leading to the development of different methods, from approaches based on classical control theory, to fully end-to-end machine learning systems, leveraging advances in computer vision, supervised and reinforcement learning.

This work proposes a grasping pipeline with a visual-servoing main component, used to precisely control the end-effector of a robotic arm towards the target object using visual feedback. A depth-camera, fixed to a mobile robot's head, observes both the object and the end-effector and estimates their positions using different methods. These camera-frame positions are used to calculate an error value which is independent from calibration of both the camera and the arm's joints. A proportional control law outputs the required end-effector velocity to minimize the error, and the arm is actuated using differential kinematics.

The target object's position is obtained by combining the bounding-box output by a convolutional neural network for object detection, with depth information from the center of the bounding-box. The end-effector is located by tracking AR tags placed around the arm wrist.

A complete grasping pipeline is developed which obtains the scene's octomap, moves the arm to a pregrasp pose where the end-effector is visible to the camera, and activates visual servo control.

The system successfully grasps several household objects in different positions, obtaining good results in a grasping benchmark scenario to be used in European Robotics League competitions.

Keywords: Robotics, Manipulation, Visual-Servoing

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Tables xiii
List of Figures xv
Glossary xvii

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement and Goals 2
1.3 Contributions 2
1.4 Document Structure 3

2 Related Work **5**
2.1 Robotic Grasping and Visual-Servoing 5
2.2 Object Grasping using Machine Learning 6
2.2.1 Supervised Learning 7
2.2.2 Reinforcement Learning 8

3 Background **11**
3.1 Robot Kinematics 11
3.1.1 Transformation Matrices and Direct Kinematics 11
3.1.2 Inverse Kinematics 12
3.1.3 Differential Kinematics 13
3.2 Visual-Servoing 14
3.2.1 Camera Configuration 14
3.2.2 Visual-Servoing Taxonomy 15
3.2.3 Position-Based Visual-Servoing 16
3.3 Object Detection 17
3.3.1 Neural Networks 17
3.3.2 Convolutional Neural Networks 19
3.3.3 YOLO Object Detection 19

4 Robot Platform Integration **23**
4.1 Hardware Modifications . . . 23
4.1.1 Choosing a Robotic Arm . . . 23
4.1.2 Installing the Arm . . . 24
4.2 Software Integration . . . 24
4.2.1 URDF . . . 25
4.2.2 MoveIt! Configuration Package . . . 26
4.2.3 Integration with MBot Class . . . 26

5 Grasping System Implementation **29**
5.1 Architecture . . . 29
5.2 Target Position Estimation . . . 30
5.3 End-Effector Pose Estimation . . . 30
5.4 Pregrasp Pose Selection . . . 31
5.5 Visual Servo Control . . . 32
5.6 Complete Pipeline Implementation . . . 33

6 Results and Discussion **35**
6.1 Experimental Setup . . . 35
6.2 Results . . . 37
6.2.1 Grasping Benchmark . . . 37
6.2.2 Data Analysis . . . 37
6.2.3 Error Tolerance Tests . . . 39
6.3 Discussion . . . 40

7 Conclusions **43**
7.1 Achievements . . . 43
7.2 System Limitations . . . 43
7.3 Future Work . . . 44

Bibliography **45**

A New MBot parts **49**
A.1 Side-plates . . . 49
A.2 Mounting piece . . . 50
A.3 Arm assembly . . . 51
A.4 Boost converter and 3D-printed pocket . . . 51

List of Tables

3.1 Taxonomy of visual-servoing types 15

6.1 Grasp performance by table position. Aggregate of all 5 objects. 37
6.2 Grasp performance by object 37

List of Figures

1.1 Three versions of MBot 2

2.1 One of the earliest neural networks used for visual-servoing [10]. **(a)** and **(b)** show the 4 features in the current and target positions, respectively. **(c)** shows the neural network architecture: the inputs are given by the differences in feature positions ($\Delta x_A = x_A - x'_A$ and so on) and the outputs are the required motions for each of the arm's joints. 7

2.2 Two-stage grasp detection process used in [13]: a smaller neural network searches for candidate grasps, and a larger network selects the best grasp pose. 8

2.3 QT-Opt: Seven identical robots autonomously collect grasping episodes [17] 9

3.1 Representation of a point P in different coordinate frames 11

3.2 Coordinate transformations in an open kinematic chain [22] 12

3.3 Camera configurations. **Left:** eye-in-hand. **Right:** eye-to-hand. On the latter, T_e^c needs to be computed by the vision system 15

3.4 Dynamic visual-servoing. **Left:** position-based. **Right:** image-based. 16

3.5 Classic visual-servoing. **Left:** position-based. **Right:** image-based. 16

3.6 Architecture of a single neuron in a neural network [26] 18

3.7 Common activation functions 18

3.8 Example of a neural network with only one hidden layer. This network's layers are *fully connected*: all neurons connect to all neurons in the next layer [27] 18

3.9 Pooling operation. Left shows the downsampling effect. Right shows how it's achieved through max-pooling, the most common method [27] 19

3.10 Full CNN architecture: convolutional layers interspersed with pooling layers produce a compact representation of high-level image features. The feature matrix is flattened, and finally one or more fully connected layers are placed. [29] 20

3.11 YOLO model predictions: bounding boxes and class probabilities are combined to obtain the final result: object labels and their bounding boxes [30] 20

4.1 **Left:** Cyton Gamma 1500, previously used in MBot. **Right:** Kinova Gen2, the arm used in our work. 23

4.2 Grasping system prototype using a simulated Kinova Gen2 arm in the Gazebo simulator . 24

4.3 Arm assembly with modified side-plate and new mounting fixture. Left to right: SolidWorks assembly, arm mounted inside the robot, final result. 25
4.4 Robot model shown in RViz, using the newly created URDF. Joints mirror the real robot. . 26
4.5 Octomap representing the environment's occupancy volume. 27

5.1 Diagram of the visual-servoing architecture. 29
5.2 Target position estimation: **Left:** YOLO bounding box recognizing the bottle. **Right:** 3D position estimation based on the bounding box's central region depth information. 31
5.3 ALVAR markers used for end-effector pose estimation 31
5.4 Grasps using the two developed pregrasp poses. 32
5.5 Steps of a grasping episode. **Left to right:** sweeping head, pregrasping, visual servo start, visual servo end and closing fingers, lifting object, moving back to resting pose after lowering object and opening fingers 34

6.1 MoCap markers placed on an object, for the benchmarking system to detect its pose . . . 36
6.2 The objects used to test the grasping system 36
6.3 3x3 grid of grasping zones 37
6.4 Plots of six different grasping episodes. The red dashed line is the time at which the pregrasp motion stops and visual servo control starts, continuing until the desired stopping distance is reached and the gripper closes. **a)** - **c)** correspond to successful grasps; **d)** and **e)** correspond to grasps on the table position furthest from the robot, causing a kinematic limit to be reached, but still gripping the object; **f)** corresponds to a failed grasp where the AR tag was never well-localized. 38
6.5 Pictures of the six grasping episodes referred in Fig. 6.4 38
6.6 Comparison of grasping with and without visual feedback, with a wrong arm measurement (8cm to the right). Visual-servoing was able to correct the error. 39
6.7 Plots comparing the versions with and without visual feedback, when a wrong 8cm translation is added to the arm. Plots correspond to three objects placed at the same table position. 40

7.1 Target localization failure caused by occlusion. 44

A.1 Original side-plates. **a)** MBot's body. **b)** Detail of right side-plate. 49
A.2 Our design: MBot body with reinforced right side-plates to support the arm, and holes for it to pass through. 50
A.3 Our design. **Left:** Outer layer. **Right:** Inner layer. 50
A.4 Mounting piece to fix the arm to the left side-plate. Front, side, and back views. 50
A.5 Assembly of the arm inside MBot's body. **a)** SolidWorks assembly. **b)** Real assembly. . . . 51
A.6 PULS CD5.243 boost converter. 52
A.7 SolidWorks model of the pocket holder. 52
A.8 Converter inside MBot, held by the 3D-printed pocket. 52

Glossary

ANN Artificial Neural Network

AR Augmented Reality

CAD Computer-Aided Design

CNC Computer Numerical Control

CNN Convolutional Neural Network

COCO Common Objects in Context

DoF Degrees of Freedom

ERL European Robotics League

IBVS Image-Based Visual-Servoing

IK Inverse Kinematics

IOU Intersection Over Union

MoCap Motion Capture

OMPL Open Motion Planning Library

PBVS Position-Based Visual-Servoing

ROS Robot Operating System

RRT Rapidly-exploring Random Tree

RSBB Referee, Scoring and Benchmarking Box

RViz Robot Visualization

SRDF Semantic Robot Description Format

SVD Singular Value Decomposition

URDF Unified Robot Description Format

VS Visual-Servoing

XML Extensible Markup Language

YOLO You Only Look Once

Chapter 1

Introduction

1.1 Motivation

The success of robotics in the industrial field is undeniable - robots are now an essential tool for many industries due to their role in improving accuracy, repeatability, reliability and efficiency. The majority of industrial robots are fixed manipulators, positioned in a manufacturing line and repeatedly performing a specific task. Building on this success, researchers have strived to deploy robots outside of these well-controlled spaces, using them to perform more complex tasks in dynamic environments.

Ever since robotics entered the collective imagination, the public has wished for robots that can help people in their everyday tasks – the 1960s cartoon ”The Jetsons” features Rosey, a robot maid that can cook, clean the house and play with children. Though not as advanced, simple robots have been built to perform household tasks such as vacuuming the floor, loading the dishwasher, and folding clothes, with mixed results.

There are active research efforts on developing more advanced domestic robots to assist the elderly and people with low mobility in their household chores, interacting with users and keeping them company. These tasks are harder to achieve, as they require a combination of capabilities such as indoors navigation, speech recognition and understanding, computer vision, object grasping, action planning, etc.

The SocRob@Home team at ISR focuses on these research problems, applying them on the MONARCH MBot robot (Fig. 1.1). Originally developed for the MONARCH project [1] with the goal of helping hospitalized children by playing games with them, it is now used in domestic robot competitions, where its functionalities are put to the test in a realistic home setting.

This work focuses on improving the grasping and manipulation functionalities on the MBot, developing a new system to grasp several household objects in a way that is less prone to calibration errors, with the purpose of achieving better results in benchmarks and competition challenges.

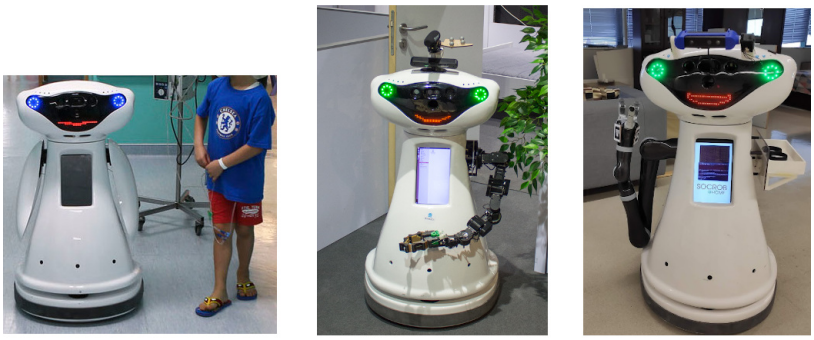


Figure 1.1: Three versions of MBot

1.2 Problem Statement and Goals

To make domestic robots be able to pick up and move objects in the home, several steps must be achieved: a) locating the object to grasp, b) planning the arm's trajectory, avoiding collisions with furniture or other objects, c) precisely placing the gripper relative to the object, d) using a grasp pose such that the object can be firmly picked up and manipulated.

Regarding point a), traditional object grasping approaches rely on placing artificial features on objects in order to locate them. Since this is unfeasible in domestic settings, our system must recognize household objects only from their image and surrounding context.

Other common shortcomings of grasping systems in domestic settings are related to c) — wrong estimation of the object's position relative to the gripper, failing to line up the arm with the object; and d) — choosing an incorrect grasp pose, such that when the gripper closes it doesn't grasp the object securely.

The goal of our system is to consistently grasp several objects, commonly used in competitions. The pose of the objects varies — in the challenges they are placed on top of a table or counter, in a random position. While we partly address the grasp selection problem, our main focus is to accurately reach a given grasping pose, near the target object.

Additionally, it is our goal to integrate these grasping functionalities in the existing ROS ecosystem of the SocRob team, such that other members can easily configure and extend the system.

1.3 Contributions

To improve the hardware capabilities of the MBot platform, we installed a new robotic arm, designing and assembling new parts to accommodate it. We integrated the arm and its drivers with the existing MBot software.

We developed a novel grasping system that can detect several household objects using supervised learning, and obtain their position using depth-sensor data. After moving the arm to a pregrasp pose, position-based visual-servoing is used to guide the gripper to the desired position, using visual markers on the gripper to track it. Since both the target and gripper positions are described in the camera frame, the servoing error is not affected by either camera or joint calibration errors.

We implemented our solution as a ROS package, added it to the MBot repository, and updated existing MBot packages to use the new grasping capabilities.

1.4 Document Structure

This thesis is structured as follows:

Chapter 3 presents the theoretical background of several techniques used in this work. Firstly, some kinematics concepts are introduced and our chosen mathematical notation is defined. Then, the main principles and variants of visual-servoing methods are outlined. Finally, some object detection techniques are described — with focus on neural networks such as the one we used to recognize the target object.

Chapter 2 presents some related research on robotic grasping. We start by mentioning traditional visual-servoing approaches, and move on to describe the state of the art on learned grasping systems.

Chapter 4 describes the integration work of installing a new robotic arm on the MBot robot, including choosing an appropriate manipulator, prototyping it in simulation, designing and assembling new robot parts, and integrating the software so that the arm can be recognized and controlled.

Chapter 5 presents our implementation of the grasping system. Firstly, the overall architecture is described. Then, the several necessary features are specified: detecting target and gripper positions, selecting a pregrasp pose, calculating the error and moving the arm using visual-servoing. Finally, the complete grasping pipeline is detailed, including how it was integrated with existing MBot software used by the SocRob team.

Chapter 6 presents the experimental setup used, results achieved and their discussion.

Chapter 7 presents the conclusions of this work, mentioning its limitations and proposing future work possibilities.

Chapter 2

Related Work

In this chapter, we analyze existing research work related to this thesis. Section 2.1 introduces robot grasping research, detailing initial uses of vision systems for grasping, and explaining closed-loop visual control — visual-servoing. Section 2.2 explores machine learning-based techniques for grasping tasks, including supervised learning and reinforcement learning approaches.

2.1 Robotic Grasping and Visual-Servoing

Grasping, manipulating and transporting objects are some of the main tasks robots are used for. The first industrial robot – Unimate – was used in the automotive industry to assemble and weld car parts. The robotic arm was programmed by storing a sequence of poses, set using manual controls. In operation, the robot would step through the poses by interpolating their joint coordinates [2].

While still used in industrial robots, this point-to-point control scheme is limited: Since the manipulator moves through static points, it will fail to reach an object that is slightly out of place. Furthermore, since the trajectory is always the same (obtained by interpolating the joint angles through time) the arm won't avoid unexpected obstacles in its way.

To address these limitations, vision-based systems were developed. McCarthy et al. describe one of the first ones [3], where a camera is used to calculate the position of a cube, obtaining the target pose for the arm's end-effector. Additionally, an occupancy model of the space is obtained from image data, making obstacle avoidance possible.

Although it enables dynamic tasks, this approach of combining vision and manipulation was still open-loop: image data is used to obtain the target coordinates fed to the manipulator's controller, which then operates separately from the visual system. The accuracy of this method depends directly on the accuracy and calibration quality of the camera and the manipulator [4].

To increase accuracy, an alternative is to establish a visual-feedback control loop, where the visual system is able to compute the position of the manipulator and correct it to the desired position. Shirai and Inoue developed an early visual-feedback system [5] that continuously recognizes the pose of a cube in the manipulator's hand, calculates the difference to the desired pose and moves the manipulator

to compensate. The block can then be precisely placed inside a box.

Subsequent projects used the vision system to provide real-time closed-loop control of a robot end-effector, making the system robust to kinematic calibration errors. This approach came to be known as *visual-servoing*.

An initial mention of the term appears in Hill and Park's work [6], in which a camera is attached to a manipulator's end-effector. The image is processed to obtain the position and orientation of the target in camera frame. Using a fixed transformation (camera to end-effector) the target is described in the end-effector's reference frame. The system uses this information to guide the end-effector to the target, considering the kinematic model of the robotic arm and its current joint positions.

These early visual-servoing systems were position-based (PBVS): visual features are extracted from the image, and a geometric model of the target object is used to estimate its 3D pose. Features are often defined as the position, orientation and size of markers placed on the target. 3D pose can be calculated from at least 3 points, and the intrinsic calibration parameters of the camera must be known [7].

Later on, the image-based visual-servoing (IBVS) technique was developed: instead of doing 3D pose estimation, image features are directly fed into a control function that outputs the end-effector Cartesian velocity. For this, a *feature-Jacobian* matrix must be defined, relating the change in feature space to the desired change in end-effector pose. IBVS doesn't require computing the geometric model of the object, improving performance by making control more direct. However, choosing visual features and defining a feature-Jacobian that behaves well for multiple poses is challenging, especially as the number of degrees-of-freedom increases [8].

Recently a new approach was introduced called Direct Visual-Servoing (DVS), which considers the whole image as an input for the control system, removing the need to add artificial image features. Collewet and Marchand propose *photometric visual-servoing*, an IBVS system that uses the luminance (intensity) of the image's pixels as the only required features [9]. An interaction matrix is defined, function of a desired image, which relates the observed image luminance to the end-effector velocity leading the arm to approach the desired pose.

2.2 Object Grasping using Machine Learning

Standard visual-servoing approaches require application-specific engineering: manually-designed image features, engineered feature-Jacobian matrix (for IBVS), and *a priori* knowledge of the target object's 3D model (for PBVS). Applying visual-servoing for grasping purposes also traditionally requires setting a predefined grasp pose (or grasping points) on the object. More recently, machine learning methods have been applied to object grasping to make systems more general. Experiments have been developed using different types of learning for visual-servoing [10, 11], 3D pose estimation [12] and grasp pose detection [13–15]. End-to-end learned systems have also been developed [16, 17], using reinforcement learning to train an agent that performs the full grasping task based only on input images.

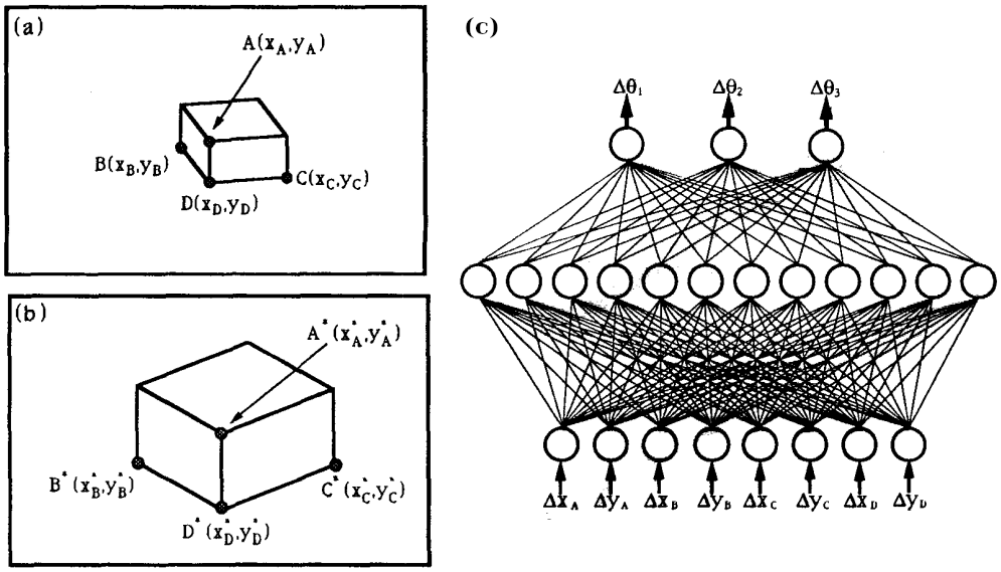


Figure 2.1: One of the earliest neural networks used for visual-servoing [10]. **(a)** and **(b)** show the 4 features in the current and target positions, respectively. **(c)** shows the neural network architecture: the inputs are given by the differences in feature positions ($\Delta x_A = x_A - x'_A$ and so on) and the outputs are the required motions for each of the arm's joints.

2.2.1 Supervised Learning

The IBVS method defines a control law which is a function of image features. These artificial features must be engineered, as well as the corresponding feature-Jacobian — the interaction matrix relating the features to desired joint velocities. Hashimoto et. al used an artificial neural network that learns the nonlinear relation between feature position changes and joint angle changes [10]. Their approach still requires an image processing step to locate four artificial features, but the feature-Jacobian is learned by the neural network. A simulated robotic arm with 3 degrees-of-freedom and a camera mounted to it (eye-in-hand) is used. In training, the arm is put in a random position, and its joint angles are recorded along with the observed image features. Then the arm is displaced a random amount, and the same information is recorded. These two positions produce a training example: the 2D difference between image feature positions is the input, and the resulting output is the difference in the 3 joint angles. Several random displacements are performed, so that there are enough examples to train the network. For execution, the feature positions observed by the camera are compared to the desired positions, and the difference is fed into the neural network, which outputs the required joint angle changes to line up the object's features with their desired positions. The network used is a fully-connected multi-layer perceptron, with 8 input neurons (x and y position deltas of the 4 visual features), a hidden layer with 12 neurons, and 3 output neurons (the angle deltas for the 3 arm actuators). Fig. 2.1 illustrates the architecture.

As previously mentioned, Direct Visual-Servoing allows for the full image to be used as an input, instead of using extracted features. In initial DVS systems [9, 18] the authors define an interaction matrix relating image values with movements of either the end-effector or arm joints. This matrix often has a small convergence domain, i.e. some starting positions will never lead to the desired end-effector

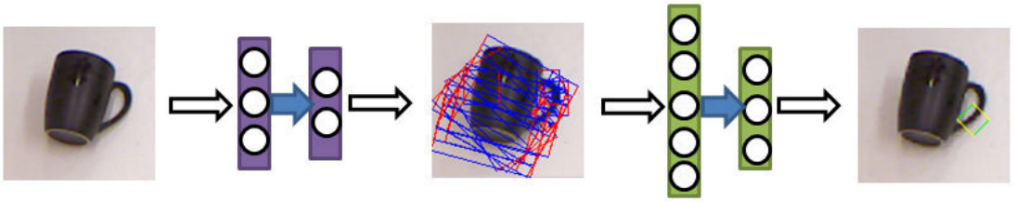


Figure 2.2: Two-stage grasp detection process used in [13]: a smaller neural network searches for candidate grasps, and a larger network selects the best grasp pose.

alignment. To improve this, Bateux et al. propose a CNN architecture that given an image as input, returns the pose (position and orientation) of the manipulator's end-effector in an eye-in-hand system [11]. This removes the requirement of manually defining an interaction matrix, and improves the system's stability, since it basically turns the control law into a PBVS one (offering global asymptotic stability [7]). The authors repurposed a pre-trained CNN, keeping the initial layers as feature extractors, and fine-tuning the final layers for outputting the 3D pose. The network is trained using simulated data in a process that generates many views of the scene, recording the corresponding 3D end-effector pose and the image from that viewpoint, to serve as training examples. This process also generates lighting variations and image occlusions to augment the dataset, making the network less sensitive to noise. A scene-agnostic version of the network is also developed, taking two images as input and outputting the end-effector displacement between them. The system is validated on a 6 DoF robot.

Lenz et al. developed a Deep-Learning algorithm to detect the best grasp pose given an object's image and depth data, without needing a 3D model [13]. The RGB-D data serves as input to a deep neural network that scores potential grasps in the image and outputs a small set of candidate grasps, which are in turn analyzed by a second, larger network that uses more features to detect the best grasp pose (see Fig. 2.2). The detection achieves good performance, and a Baxter robot performs several successful grasps using the algorithm.

2.2.2 Reinforcement Learning

Reinforcement Learning (RL) is widely used in robotics, as it allows robots to autonomously discover optimal behaviors through trial-and-error interactions with the environment. While supervised learning methods require large datasets of labeled examples, in RL the designer only needs to define a reward function for the agent to maximize. A large amount of experience data still needs to be acquired, but it's gathered by the agent while interacting with the environment. To accelerate experience gathering, simulators are often used to replicate the environment and the robotic agent.

Kohl and Stone [19] used reinforcement learning to optimize the gait of the Sony AIBO quadruped robot. The agent was trained using the policy gradient RL technique with a reward function which only considers forward speed. Experience is gathered by three AIBO robots walking simultaneously, starting with an initial policy which mimics the robot's default walking mechanism. After only three hours of learning, the achieved policy makes the robot walk faster than the default stride, and also faster than other hand-coded approaches.

Figure 2.3: QT-Opt: Seven identical robots autonomously collect grasping episodes [17]

More recently, deep reinforcement learning (DRL) was introduced, using deep neural networks (often CNNs) for representing states and policies. This allows learning even in the case of large state and action spaces. DRL has been used to learn locomotion gaits of quadruped [20] and simulated humanoid [21] robots.

Kalashnikov et al. introduced a distributed deep reinforcement learning framework – QT-Opt – for grasping and manipulation of novel objects [17]. Their scalable architecture can process experience data from multiple robots (with identical control and kinematic structures, see fig. 2.3), store it in a distributed replay buffer database, and parallelize training across multiple GPUs, using a Q-learning based algorithm to update weights. This work achieves a very high success rate for grasps on unseen objects, in a fully end-to-end approach from image pixels to robot actions, with no knowledge about objects, physics, or robot motion planning. A large amount of experience data is required to make the system converge: 7 robotic arms were used to obtain data from 580k real-world grasps.

Chapter 3

Background

This chapter presents the theoretical background for the main methods and algorithms used in this work. Section 3.1 introduces robot kinematics notation and concepts, including inverse and differential kinematics, used for motion planning and visual-servoing. Section 3.2 presents the visual-servoing algorithm and its variations, used in the work to achieve a precise grasp guided by the vision system. Finally, section 3.3 explains deep neural networks and their use for object detection, which we apply for detecting the grasp target pose.

3.1 Robot Kinematics

3.1.1 Transformation Matrices and Direct Kinematics

In robotics it is common to deal with multiple *frames of reference*. In the example of a robot with a camera, it's useful to translate the spatial coordinates of a point from the camera frame to the robot frame.

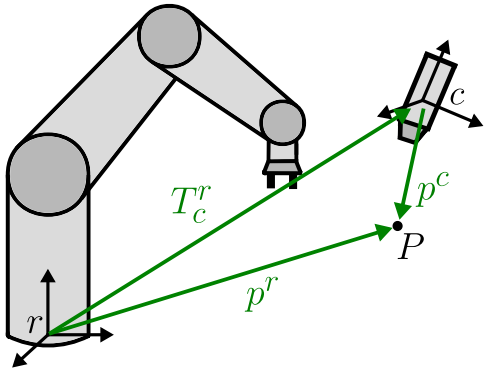


Figure 3.1: Representation of a point P in different coordinate frames

It is possible to represent a mathematical relationship between two frames by using a *homogeneous transformation matrix*. In Fig. 3.1, if p^c are the Cartesian coordinates of a point P in the camera frame, and we wish to obtain p^r, the coordinates of P in the robot frame, we can do so by using T_c^r, the homogeneous transformation matrix from c to r:

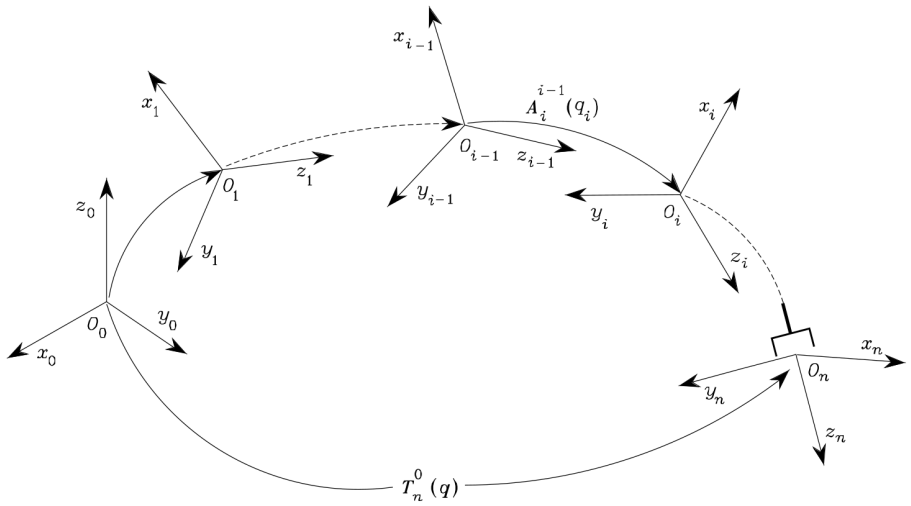


Figure 3.2: Coordinate transformations in an open kinematic chain [22]

$$p^r = T_c^r p^c \tag{3.1}$$

This notation is used in the rest of this work.

A homogeneous transformation matrix takes the form:

$$A_1^0 = \begin{bmatrix} R_1^0 & o_1^0 \\ 0^T & 1 \end{bmatrix} \tag{3.2}$$

where R_1^0 is the rotation matrix of frame 1 with respect to frame 0 and o_1^0 is the translation vector from the origin of frame 0 to the origin of frame 1 [22].

Frame transformations are also used for manipulator kinematics. In *direct* or *forward kinematics* we wish to compute the pose of the end-effector as a function of the joint angles q. For a manipulator composed by n joints and $n+1$ links, we can attach a coordinate frame to each link as pictured in Fig. 3.2. The transformation describing the pose of link n (the end-effector link in the picture) with respect to link 0 (robot base) is given by the kinematic chain:

$$T_n^0(q) = A_1^0(q1)A_2^1(q2)\cdots A_n^{n-1}(qn) \tag{3.3}$$

where A_i^{i-1} is the homogeneous transformation matrix between two consecutive link frames, function of qi, the current angle of joint i connecting the links.

3.1.2 Inverse Kinematics

For manipulation tasks it is necessary to transform end-effector poses and motions into joint space values and velocities. This is the *inverse kinematics* problem.

Regarding the direct kinematics equation (3.3), given a set of joint angles there is always one solution for the end-effector pose. The inverse kinematics problem is more complex: given an end-effector pose

there can be multiple solutions, infinite solutions, or no solutions for the joint values. Calculating all exact solutions is challenging, as they depend on the degrees-of-freedom, mechanical joint limits and dexterous workspace of the manipulator [23].

By analytically inverting the direct kinematics equations, all solution branches can be enumerated. But this only works when the number of constraints is equal to the degrees-of-freedom of the robot. For example, in a manipulator with 6 degrees-of-freedom and no joint limits, placing its end-effector in 3D space (6 constraints: 3 positional and 3 rotational), generally up to 16 solutions can be found. By adding one more joint (7 DoF arm), the number of solutions becomes infinite, and no analytical solution exists. While it is possible to fix one joint and analytically solve for the others, numerical techniques exist that can find approximate solutions for any kind of manipulator.

3.1.3 Differential Kinematics

Differential kinematics is used to obtain the relationship between joint velocities and the corresponding end-effector velocity. This is done by calculating the Jacobian matrix, which depends on the manipulator configuration. End-effector velocity v_e is given by

$$v_e = J(q)\dot{q} \tag{3.4}$$

where $\dot{q}$ is the vector of linear joint velocities, and $J(q)$ is the Jacobian matrix, dependent on joint angles q.

The Jacobian matrix can be obtained based on direct kinematics relations, and is expressed in the following way:

$$J = \begin{bmatrix} \mathcal{J}_{P1} & \dots & \mathcal{J}_{Pn} \\ \mathcal{J}_{O1} & \dots & \mathcal{J}_{On} \end{bmatrix} \tag{3.5}$$

where

$$\begin{bmatrix} \mathcal{J}_{Pi} \\ \mathcal{J}_{Oi} \end{bmatrix} = \begin{cases} \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix} & \text{for a prismatic joint} \\ \begin{bmatrix} z_{i-1} \times (p_e - p_{i-1}) \\ z_{i-1} \end{bmatrix} & \text{for a revolute joint} \end{cases} \tag{3.6}$$

Vectors z_{i-1}, p_e and p_{i-1} are all functions of the joint angles, and obtained from different columns of the transformation matrices described in (3.3).

Determining the joint velocities required for a certain end-effector velocity is possible by inverting (3.4):

$$\dot{q} = J^{-1}(q)v_e \tag{3.7}$$

This is the *inverse differential kinematics* problem. Since J may not be square or invertible, obtaining J^{-1} is not always possible. Instead, we can calculate the *Moore-Penrose inverse* (or *pseudoinverse*) $J^{\dagger}$,

which is defined and unique for all numerical matrices. It can be obtained in several ways, but the most common computational method is using *singular value decomposition*:

$$J = UDV^T, J^\dagger = VD^\dagger U^T \tag{3.8}$$

Given that J is $m \times n$, U is an $m \times m$ matrix of left-singular vectors, V is an $n \times n$ matrix of right-singular vectors, D is an $m \times n$ diagonal matrix, and $D^\dagger$ is $n \times m$, formed from D by taking the reciprocal of the non-zero elements and keeping the zeros.

Computing $J^\dagger$ this way, and using it in place of J^{-1} in equation (3.7) provides a computational method to obtain the joint velocities required for a desired end-effector velocity.

This method can be extended to solve inverse kinematics: instead of the desired end-effector velocity v_e, we can use direct kinematics to compute $\Delta p = p(q_0 + \Delta q) - p(q_0)$, the change in end-effector position given the current joint angle changes Δq. Δq can be iteratively improved using the Newton-Raphson method, minimizing an error function that measures distance to the desired end-effector position ($error = ||p(q_0 + \Delta q) - p_{desired}||$) [22].

3.2 Visual-Servoing

Visual-servoing is a method of robot control which uses visual feedback to control the robot's actuators continually, in closed-loop. Computer vision techniques are used to extract image features from the camera and identify the position of the target object in relation to the manipulator's gripper.

Over time, several visual-servoing approaches have been developed, which can be characterized based on their camera configuration and control architecture. Below we present and explain different types of visual-servoing.

3.2.1 Camera Configuration

There are two typical configurations for the camera in visual-servoing systems: end-effector mounted, or fixed in the workspace [4]. Fig. 3.3 shows a comparison between the two.

In the end-effector mounted configuration (also called *eye-in-hand*) there is a fixed transformation between the camera and the end-effector's pose, making it possible to estimate the target's position in the end-effector frame, which is the frame used for control. This estimation is simpler, but the camera's view is limited by the current orientation of the arm.

In the configuration where the camera is fixed in the workspace (*eye-to-hand*) it's necessary to obtain the pose of both the end-effector and the target, since their movement is independent. This approach requires additional work to identify the end-effector's pose in the image, but has the advantage of providing a panoramic view of the workspace.

There have also been experiments on multi-camera systems which implement hybrid *eye-in-hand* / *eye-to-hand* approaches, merging information from both camera configurations [24].

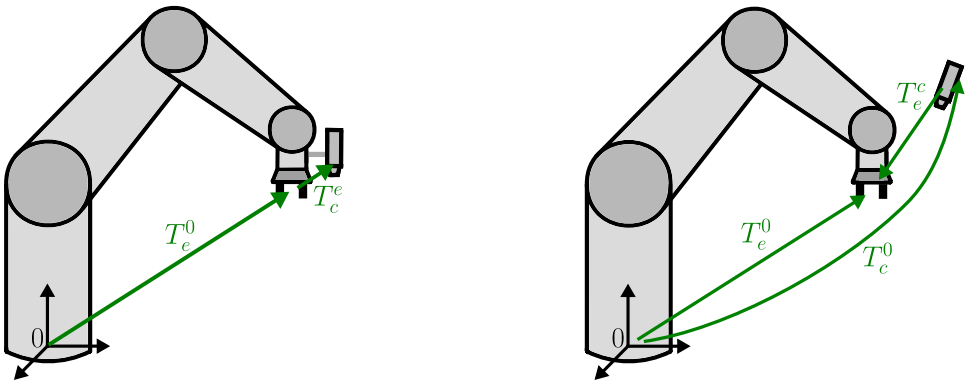


Figure 3.3: Camera configurations. **Left:** eye-in-hand. **Right:** eye-to-hand. On the latter, T_e^c needs to be computed by the vision system

3.2.2 Visual-Servoing Taxonomy

Visual-servoing systems evolved in different ways regarding their architecture. Sanderson and Weiss [25] describe a taxonomy of visual-servoing systems including four categories, based on the way they answer two questions:

1. Does the system apply joint inputs (angles and/or velocities) directly, or does it use joint sensor feedback for stabilization?
2. Is the error signal defined directly in terms of image features, or in 3D workspace frame using pose estimation?

Table 3.1 relates visual-servoing categories with the way they solve these problems.

		Error signal	
		Computed 3D pose	Defined in image space
Joint inputs	Stabilized by joint feedback	Dynamic position-based	Dynamic image-based
	Directly applied	Classic position-based	Classic image-based

Table 3.1: Taxonomy of visual-servoing types

In *classic visual-servoing* (also called simply *visual-servoing* or *direct visual-servoing*, not to confuse with the more recent DVS approach mentioned in the previous chapter) the system directly applies joint inputs to the manipulator's motors. *Dynamic visual-servoing* systems (also called *dynamic look-and-move*) make use of joint feedback sensors to stabilize the robot. Currently most systems use the latter approach, taking advantage of modern robot controllers which accept Cartesian position and velocity commands, and connect to joint angle and torque sensors for stabilization.

Image-based servoing systems (IBVS) compute the error (difference between end-effector and target poses) directly in image space, and the manipulator control law is a function of the image feature positions. Since it operates in the feature space of a 2D image, there is a loss of dimensional information which can cause localization issues. In fact IBVS control has been shown to be asymptotically stable

only locally [7], that is, the system can be attracted to a local minimum that doesn't correspond to the desired configuration.

Position-based systems (PBVS) estimate the 3D pose of the target, using it to compute a Cartesian control law for the manipulator. PBVS control classically requires solving the *3D localization problem*, which adds a performance cost and can introduce errors. However, PBVS offers better stability guarantees than IBVS: it is shown to be globally asymptotically stable [7].

Figs. 3.4 and 3.5 illustrate the internal architectures of the four visual-servoing categories. The *eye-to-hand* configuration is depicted, but the *eye-in-hand* one would also be valid, since these categories are independent of camera configuration.

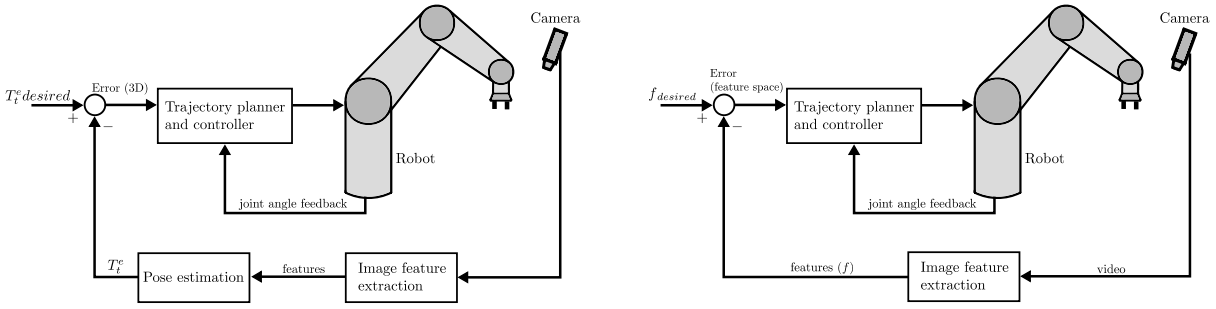


Figure 3.4: Dynamic visual-servoing. **Left:** position-based. **Right:** image-based.

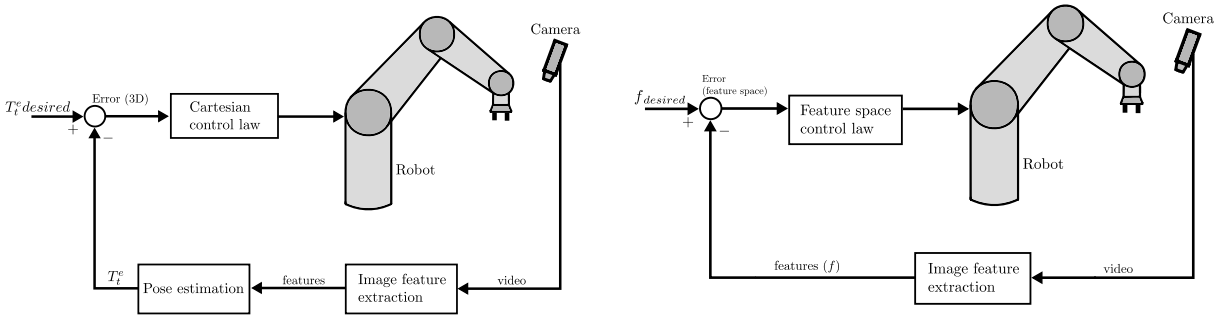


Figure 3.5: Classic visual-servoing. **Left:** position-based. **Right:** image-based.

3.2.3 Position-Based Visual-Servoing

In position-based control (the method used in this work), image features are used to compute an estimate of the target pose, which is then used for control. The control function is therefore separated from the pose estimation problem. In the case of a fixed camera (eye-to-hand, as used in our system) the kinematic error function is given by [4]:

$$E(T_e^0, h^e, g^0) = T_e^0 h^e - g^0 \tag{3.9}$$

T_e^0 is the transformation matrix from the end-effector frame (e) to the arm's root frame (0), which is the variable we can control by actuating the arm (the matrix is function of the joint angles). h^e are the coordinates, in end-effector frame, of a point on the arm (the *tool* or *hand*) which we want to place in a

fixed goal position g^0.

We wish to apply a linear velocity u^0 to the end-effector to minimize the error above. Open-loop positioning can be achieved by applying the proportional control law

$$u^0 = -k\ (\hat{T}_e^0 h^e - \hat{T}_c^0 \hat{g}^c) \tag{3.10}$$

where $\hat{T}_e^0$ is shown as an estimate (as it is subject to errors in the arm's joint sensors), $\hat{T}_c^0$ is the estimated transformation matrix from the camera frame to the root frame (usually based on fixed transforms from manual measurements) and $\hat{g}^c$ is the estimated pose of the goal in camera frame, given by the vision system. $k > 0$ is a proportional feedback gain.

Because they are estimated, errors in $\hat{T}_e^0$, $\hat{T}_c^0$ or $\hat{g}^c$ will lead to end-effector positioning errors.

The control loop can be closed by continuously observing h^e (the hand) and estimating its position. Doing this, (3.10) turns into

$$u^0 = -k\ \hat{T}_c^0(\hat{h}^c - \hat{g}^c) \tag{3.11}$$

where $\hat{h}^c$ are the camera frame coordinates of the observed hand point, as estimated by the vision system.

In this equation u^0 doesn't depend on $\hat{T}_e^0$ anymore. Also, if $\hat{h}^c$ and $\hat{g}^c$ (dependent on the same camera calibration) are equal, then $u^0 = 0$ and equilibrium is reached, independently of errors in robot kinematics or camera calibration. This is the crucial advantage of visual-servoing algorithms.

3.3 Object Detection

To detect the object to grasp, our system employs the YOLO algorithm, which uses a convolutional neural network (CNN) for real-time object detection. This section introduces neural networks, explains CNNs, and specifies how YOLO uses them.

3.3.1 Neural Networks

A neural network is a supervised learning technique which uses a connected network of artificial neurons (also called perceptrons) for classification. For each neuron, its input vector **x** is multiplied by a weight vector **w** and a bias value b is added. The result is fed to an activation function σ which computes the neuron's output y. Fig. 3.6 illustrates this process.

Like other supervised learning techniques, given a dataset consisting of several input/target pairs $\{\mathbf{x}, t\}$ the goal is to learn a model of the relationship between **x** and t. To train the neural network is to obtain the weight vector **w** that produces a function y as close as possible to t.

Since target function t is often not a linear function of **x**, nonlinearity must be introduced in the system. This is the goal of activation function σ. Several nonlinear functions can be used with different performance characteristics, the most common being the sigmoid, $tanh$ and ReLU — see Fig. 3.7.

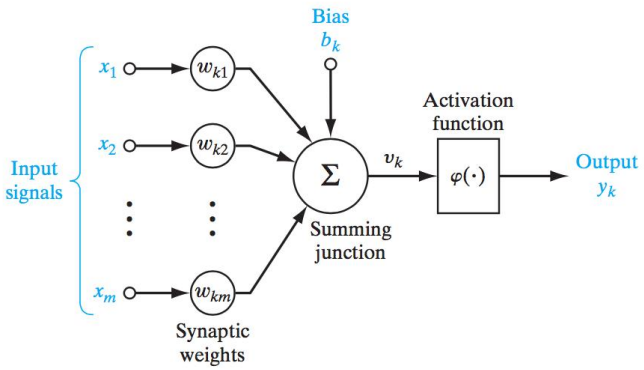


Figure 3.6: Architecture of a single neuron in a neural network [26]

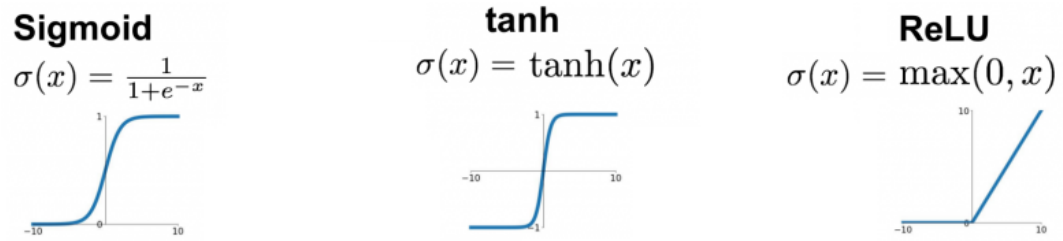


Figure 3.7: Common activation functions

Neural networks combine several neurons and group them in layers. The layer where input values enter the network is called the *input layer*. The layer which produces output values is called the *output layer*. In between them, there can be several *hidden layers* with any number of neurons (see Fig. 3.8). Several network topologies have been developed for different purposes, e.g. Convolutional Neural Networks (mostly used for computer vision tasks) which introduce convolutional layers that apply image filters, and Recurrent Neural Networks (commonly used for natural language processing) which contain loops, allowing for the network's output to be influenced by temporal sequences in the input.

To train a network, for every dataset example $\{\mathbf{x}, t\}$ an error signal is calculated:

$$e = ||t - y(\mathbf{x})||^2 \tag{3.12}$$

with $y(\mathbf{x})$ given by the network with current weight vector $\mathbf{w}$. While this corresponds to quadratic error, other error functions can be used.

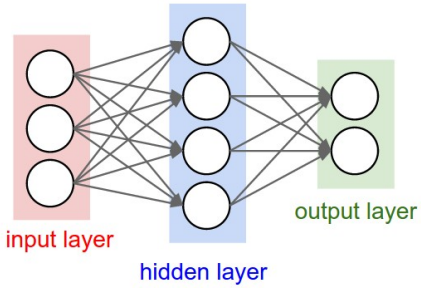


Figure 3.8: Example of a neural network with only one hidden layer. This network's layers are *fully connected*: all neurons connect to all neurons in the next layer [27]

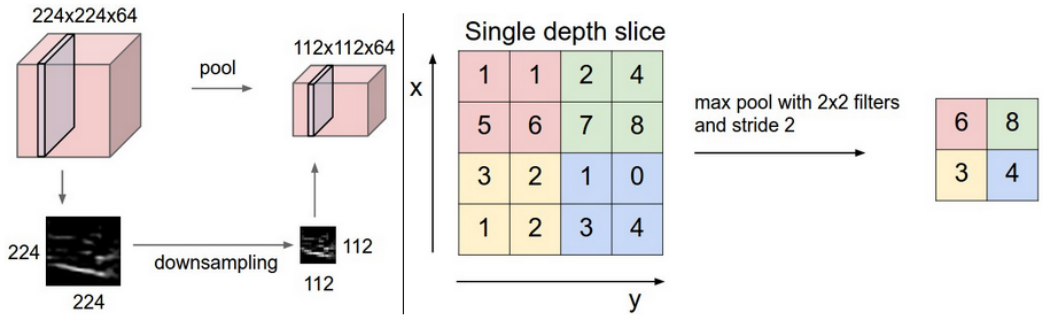


Figure 3.9: Pooling operation. Left shows the downsampling effect. Right shows how it's achieved through max-pooling, the most common method [27]

Then, the weights are adjusted using gradient descent, in a direction which minimizes the error:

$$w_{n+1} = w_n - \eta \frac{\delta e}{\delta \mathbf{w}} \tag{3.13}$$

η is the *learning rate* parameter, which can be set to a constant or be a decreasing function over the training time. The derivative of the error in respect to the weights ($\frac{\delta e}{\delta \mathbf{w}}$) can be calculated as a product of derivatives between each layer of the network, using the backpropagation algorithm [28].

Changing the weights after processing each individual input/target pair is described as *online learning*. To optimize performance, a *batch learning* algorithm can be used that processes a batch of several examples at a time, using their combined outputs and errors to update the weights.

3.3.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a type of neural network architecture, commonly used to process images. The main component of CNNs is the *convolutional layer*: it consists of a set of learnable filters, that slide (convolve) through the image space and activate when they detect visual features such as edges or special shapes. The initial convolutional layer detects simple shapes, while deeper layers detect more complex patterns.

Between consecutive convolutional layers, *pooling layers* are inserted. Their purpose is to downsample the image representation, reducing its spatial size and therefore the amount of parameters, making the network more efficient and less susceptible to noise. Pooling layers don't add learnable parameters to the network, since they apply a fixed operation on the input. Fig. 3.9 shows a pooling layer operation.

Fully connected layers are commonly placed at the end of the CNN, after the highest level feature outputs. This is done so that nonlinear combinations of these high-level features can be learned. In the case of image classification, the size of the final output vector is equal to the number of detectable classes (see Fig. 3.10).

3.3.3 YOLO Object Detection

Initially CNNs were mostly used for image *classification*, which finds the most relevant class (e.g. cat, flower, car, etc.) given an input image. Later, CNNs were adapted for object *detection*: identifying and

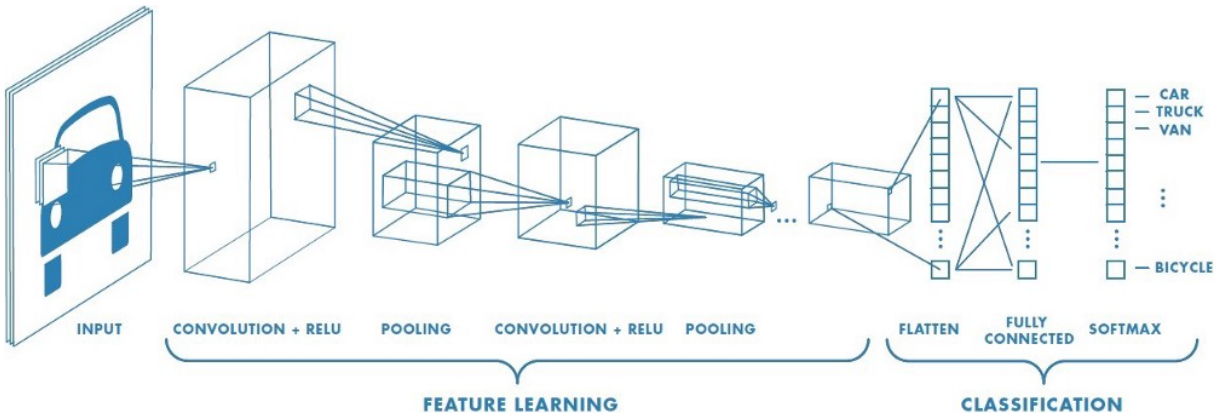


Figure 3.10: Full CNN architecture: convolutional layers interspersed with pooling layers produce a compact representation of high-level image features. The feature matrix is flattened, and finally one or more fully connected layers are placed. [29]

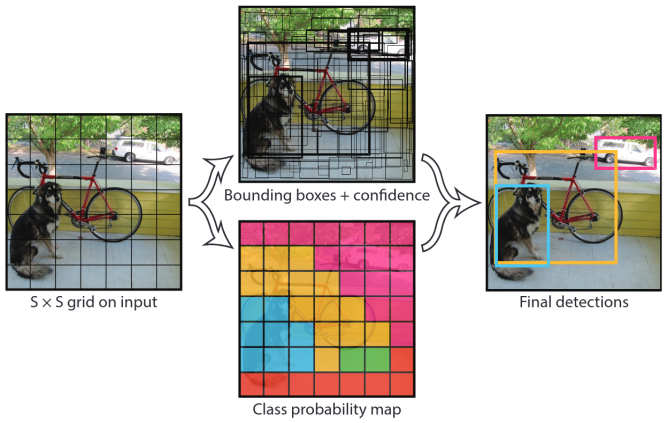


Figure 3.11: YOLO model predictions: bounding boxes and class probabilities are combined to obtain the final result: object labels and their bounding boxes [30]

locating several objects in an image. Early object detection systems were simply repurposed classifier networks, applying them at various image subregions. The location and scale of the subregions can also be learned parameters. The runtime performance of these systems is poor, since the classification process must be run many times.

YOLO (You Only Look Once) [30] is an object detection approach that uses a single CNN to predict bounding boxes and class probabilities of objects. The network runs only once to predict bounding boxes and probabilities for all classes. This improves its runtime performance, making it able to perform real-time object detection at more than 30 frames per second.

The input image is divided into an $S \times S$ grid. Each cell predicts B bounding boxes centered on that cell. Confidence scores are assigned to the boxes, reflecting how likely it is for the box to contain an object, and how accurate the box is, measured by the intersection-over-union (IOU) metric.

Each cell also predicts C class probabilities $P(\text{Class}_i|\text{Object})$, where C is the number of detectable classes. This number is independent of the number of bounding boxes B. This means it's difficult for YOLO to detect small objects that are very close or intersecting with each other.

The YOLO CNN has 24 convolutional layers, with maxpool layers in between them. 2 fully connected layers are placed at the end. The first 20 convolutional layers are pretrained on the ImageNet dataset,

4 additional layers are added to perform detection, and the model is trained based on ground-truth bounding boxes. The loss function penalizes a) classification error for cells that contain objects, and b) bounding box misalignment based on IOU.

Chapter 4

Robot Platform Integration

This chapter presents the hardware and software integration work done to improve MBot's grasping capabilities. A new robotic arm was installed on the MBot platform, and integrated with its software.

The work done in this section serves as the foundation not only for our work, but also for other manipulation systems implemented on the same MBot platform.

4.1 Hardware Modifications

4.1.1 Choosing a Robotic Arm

Since MBot is a mobile robot of medium dimensions, its robotic arm cannot be too large and heavy as it might put too much weight on the wheel-base and cripple the robot's movement and navigation. In previous robotics challenges, the SocRob team used a Robai Cyton Gamma 1500 arm — Fig. 4.1, left. While lightweight, its hardware was not optimal. The joint servos had some "play", meaning high positioning precision was not possible. Also, the angle sensors would frequently become erratic, requiring manual recalibration.

Figure 4.1: **Left:** Cyton Gamma 1500, previously used in MBot. **Right:** Kinova Gen2, the arm used in our work.

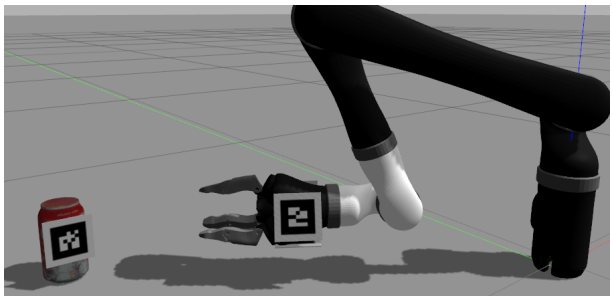


Figure 4.2: Grasping system prototype using a simulated Kinova Gen2 arm in the Gazebo simulator

As part of our effort in improving the manipulation system, we decided to remove the Cyton arm and replace it with a higher-precision one. We chose to install a Kinova Gen2 6DoF robot arm (Fig. 4.1, right) which features six integrated actuators, each one housing a brushless DC motor, a position sensor with more than one million positions per turn, torque, current and temperature sensors. Another reason for our selection was ROS support: Kinova maintain official ROS driver packages allowing for positional and velocity control of the joints and end-effector. The drivers also include visual and kinematic 3D models of the arm, and a package allowing for simulated control of the arm in Gazebo.

Before acquiring the arm, we used the `kinova_gazebo` ROS package to test it in simulation and evaluate the feasibility of grasps. We prototyped a baseline grasping system with the arm attached to the ground and a vertical base (Fig. 4.2).

4.1.2 Installing the Arm

Because of the arm's considerable length (98.5cm, while the Cyton arm has 68cm) we decided to mount the arm's base inside MBot's body, to make the whole robot more compact — see Fig. 4.3. To achieve this we modified the existing right-side plate in SolidWorks, adding a hole for the arm to pass through it. Two copies of the part were machined using 2D CNC milling, and the double layer was installed in the robot replacing the old single layer plate, adding support. We also designed a new part which attaches the arm's base to the inside of MBot's left-side plate. This part is made of acetal (also called polyoxymethylene or POM) and was machined using 3D CNC milling, according to our CAD model.

The Kinova Gen2 requires a supplied voltage of 18 to 29 Volts DC (24 Volts nominal). Since the robot's batteries only provide 12V current we installed a boost converter, held by a 3D-printed mounting piece. We added an easily accessible button switch between the battery and the boost converter, so that the arm can be quickly turned off in case of emergency.

Additional details for the parts are available in appendix A. CNC machining was performed by IST spinoff company IDMind, which also helped in design and material choices.

4.2 Software Integration

After installation, the arm was integrated with MBot's ROS packages so that its joints can be observed and actuated, allowing packages like MoveIt! to plan and execute trajectories. Common arm functions

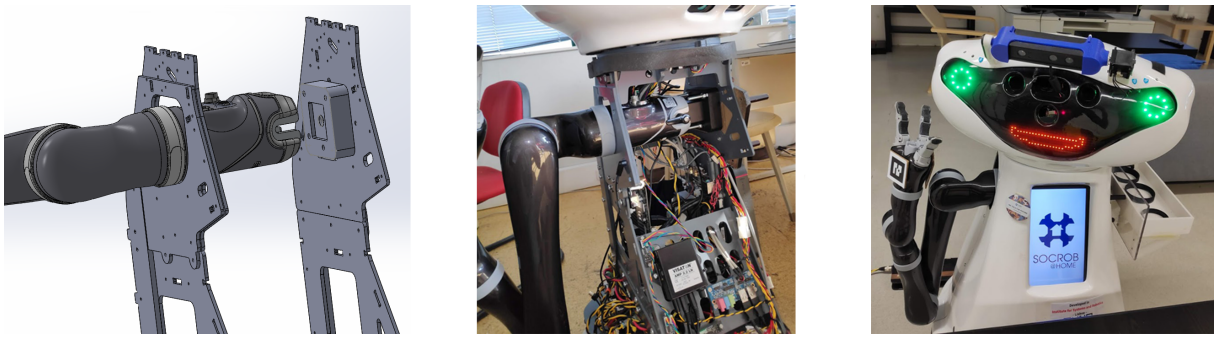


Figure 4.3: Arm assembly with modified side-plate and new mounting fixture. Left to right: SolidWorks assembly, arm mounted inside the robot, final result.

(i.e. moving to a certain pose, opening and closing the gripper) were implemented and made available for SocRob members to implement custom procedures using the arm.

4.2.1 URDF

To add the configuration of the arm's joints and links, as well as their 3D meshes, to the existing MBot model, we need to configure MBot's URDF (Unified Robot Description Format [31]) file. The existing MBot description file is imported, along with the arm's URDF available in the `kinova_description` package. The arm's `root` frame is defined, and a fixed transform (a fixed "joint" in URDF terms) from the robot's base frame (`base_link`) to `root` is added, based on the measured distance and rotation from the robot base to the arm base.

The xacro [32] tool is used to enable XML macro usage. Listing 4.1 below displays a simplified version of the `mbot_with_kinova_arm.xacro` file used in implementation.

```
<robot xmlns:xacro="http://wiki.ros.org/xacro" name="mbot">
  <xacro:include filename="$(find mbot_description)/xacro/mbot.xacro"/>
  <xacro:include filename="$(find kinova_description)/urdf/j2s6s300.xacro"/>
  <xacro:j2s6s300 base_parent="root"/>

  <joint name="mbot_base_to_arm_root" type="fixed">
    <parent link="base_link"/> <child link="root"/>
    <origin xyz="${arm_x} ${arm_y} ${arm_height}" rpy="${M_PI/2} 0 0"/>
  </joint>
</robot>
```

Listing 4.1: Xacro file connecting Kinova arm to MBot robot. `j2s6s300` refers to our arm's version: Jaco model, 6DoF, spherical wrist. `arm_x`, `arm_y` and `arm_height` are pre-measured distances.

When the arm is connected, the driver package communicates with its joint sensors to update the URDF joint states, making the robot model (observable in RViz, as shown in Fig. 4.4) mirror the actual robot joints.

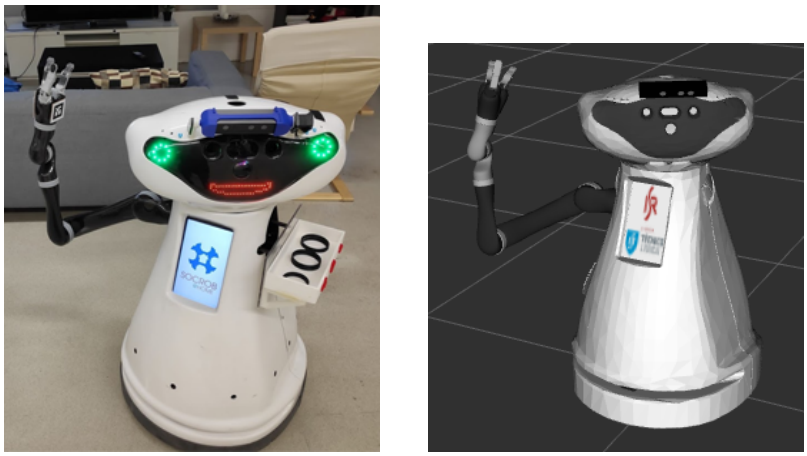


Figure 4.4: Robot model shown in RViz, using the newly created URDF. Joints mirror the real robot.

4.2.2 MoveIt! Configuration Package

The MoveIt! ROS framework [33] includes several grasping and manipulation utilities including inverse kinematics, motion planning, obstacle avoidance based on 3D perception, visualization, interactive planning, etc. Its modular design allows for different algorithms to be used interchangeably. To make use of the MoveIt! pipeline, not only in this work but for other manipulation tasks on the same platform, a MoveIt! configuration package is necessary.

To create this package we use the MoveIt! Setup Assistant. We load the URDF configured as above, select the movable joints, identify the end-effector and establish the planning group — a kinematic chain from base to end-effector, on which motion planning will be calculated. We then add some fixed robot poses, which allow the programmer to plan and move the arm to them based on the pose name. Finally we change the `point_cloud_topic` configuration parameter — we set it to the point cloud captured by the depth camera on MBot's head. This is then used to build the octomap of the scene — a 3D occupancy map used to avoid obstacles in motion planning. Fig. 4.5 shows a scene's octomap.

MoveIt! includes several algorithms for inverse kinematics. We configured the package to use the TRAC-IK solver [34], which obtains good solve rates for challenging positions while keeping computation time low, by running two solvers concurrently — one faster but worse at finding solutions, and one with higher solve rates but higher computation time — and stopping when the first solution is found.

As for the motion planning algorithm, MoveIt! can use any planner compatible with the Open Motion Planning Library (OMPL) [35]. The RRT-Connect algorithm [36] was chosen due to its efficiency, achieved by building two Randomly-exploring Random Trees (RRT), one from the start point and one from the end, incrementally advancing towards each other until they connect.

4.2.3 Integration with MBot Class

MBot Class is an important part of SocRob team's software. It is a python structure consisting of several functionality classes. Each class exposes functions related to its capability. For example, the `hri` (Human-Robot Interaction) class includes the functions `say(sentence)` and `set_emotion(emotion)`, respectively to speak a given sentence using text-to-speech, and to set the robot's eye color and mouth

Figure 4.5: Octomap representing the environment's occupancy volume.

shape according to a given emotion.

For our work we developed a `kinova_manipulation` functionality class and added it to MBot Class. It exposes several convenient arm functions, listed below:

- `set_fingers(finger1, finger2, finger3)`, `open_gripper()` and `close_gripper()`: `set_fingers()` sends a goal to the "finger_positions" ROS action server, with values to actuate each of the three fingers, from 0 to 100 — 0 fully opens, 100 fully closes the finger. `open_gripper()` and `close_gripper()` set 0 and 100 values, respectively, for all fingers.
- `go_to_pose(pose_name)`: Connects to MoveIt! to plan and execute a motion from the current pose to the given predefined pose, identified by its name in the MoveIt! configuration file.
- `get_eef_pose()`: Returns the current end-effector pose in the `base_link` frame, obtained from MoveIt! which uses forward kinematics to calculate it, based on the URDF and current joint angles.
- `move_eef_relative(x, y, z)`: Obtains the current end-effector pose using the function above, applies the translation given by the arguments, and executes a motion to the translated pose.
- `grasp_object(object_class)`: Attempts to grasp an object of the given class by executing the grasping pipeline developed in this thesis and described in the next chapter.

This integration is not only useful for the pipeline developed in this work, but can also be used for other future grasping pipelines.

Chapter 5

Grasping System Implementation

This chapter describes the development and implementation of our grasping system. Starting with the high-level visual-servoing equation, we then specify how we estimate the relevant variables, i.e. the end-effector and target object positions. We explain the proportional control scheme used, and the calculations for the joint velocities to apply at every timestep. Finally, the full grasping pipeline is laid out.

5.1 Architecture

A traditional visual-servoing control scheme was chosen, using supervised learning for the visual feedback, specifically for obtaining the target object's position. To achieve this, the YOLO CNN-based object detection algorithm is used to output a 2D bounding box over the part of the image containing the target object. By sampling image depth, the 3D position of the object's center-point is estimated.

Our system takes advantage of MBot's depth-camera, fixed to its head. This makes it an eye-to-hand system, requiring the end-effector's pose to be estimated by the vision system. To do this, three AR markers were added to the arm's wrist, and the ALVAR package is used to track their 3D position.

The error is given by the difference between the target object and end-effector positions, in camera frame. A proportional control law is used to obtain the end-effector velocity required to minimize the error. Using singular value decomposition to compute the Jacobian pseudo-inverse, the joint velocities

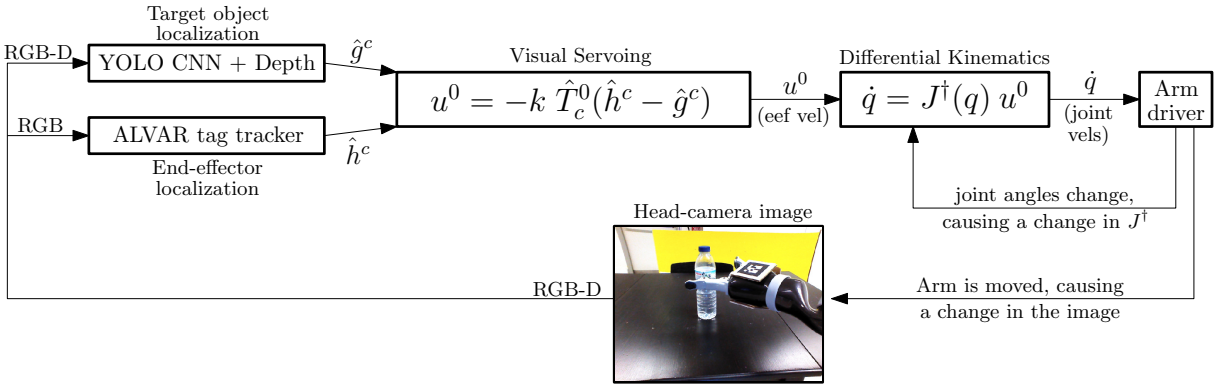


Figure 5.1: Diagram of the visual-servoing architecture.

are calculated and given as inputs to the arm's controller. Fig. 5.1 shows the diagram of the system.

As opposed to using a more end-to-end approach, our solution uses supervised learning as a "sensor" that provides the object position as an input for a classical position-based visual-servoing control algorithm (PBVS). This creates a separation of concerns between perception and control, facilitating debugging and error analysis. Assuming the target is correctly localized, the system takes advantage of the well-studied stability guarantees of PBVS [4, 7], whereas end-to-end systems must learn the control function through examples, requiring a large dataset of experimental data. Additionally, we make use of the same neural network architecture that is used for other object detection purposes in MBot, meaning a single model can be used for multiple tasks, or a grasping-specific model can be trained, maintaining the same network architecture and detection pipeline.

5.2 Target Position Estimation

As mentioned above, a supervised learning approach is used to localize the target object. The YOLO v3 CNN receives RGB images from the head-camera and outputs 2D bounding boxes around the recognized objects. This enables the system to leverage existing pre-trained models for object detection, as well as using competition-specific models trained on images of objects featured in robot competitions. During development we used a pre-trained network capable of identifying 80 common objects from the COCO dataset [37]. The network consists of 75 convolutional layers with leaky ReLU activation. Instead of using max-pool layers, downsampling is done by convolutional layers with a stride of 2 (sliding 2 pixels at a time), preventing the loss of low-level features. Bounding box detections are performed at three different image scales [38].

Since our system assumes knowledge about the type of object to grasp, only the bounding boxes corresponding to that class are considered. A region of points in the center of the bounding box is sampled, and the depth values of those points are obtained from the captured depth image. The 25^{th} percentile depth value is chosen as representative of the surface's depth, in order to eliminate outliers and points which don't correspond to the object. The depth value is denominated d.

A pinhole camera model, created from the camera calibration parameters, is used to obtain a rectified image without lens distortion. The center pixel of the bounding box is chosen and a ray is projected from the lens, intersecting the rectified pixel. This ray is represented by the unit vector $\mathbf{r} = [x\ y\ z]^T$ which points in the direction of the object's center-point. The depth value is multiplied by this vector to obtain the 3D position of the goal point in camera frame: $\hat{g}^c = d \cdot \mathbf{r}$.

5.3 End-Effector Pose Estimation

To localize the end-effector, the ALVAR ROS package was used. ALVAR is an open source library for tracking and localizing AR tags based on the size and angle of the observed visual features, and *a priori* knowledge of the marker's size and the camera's intrinsic parameters [39].

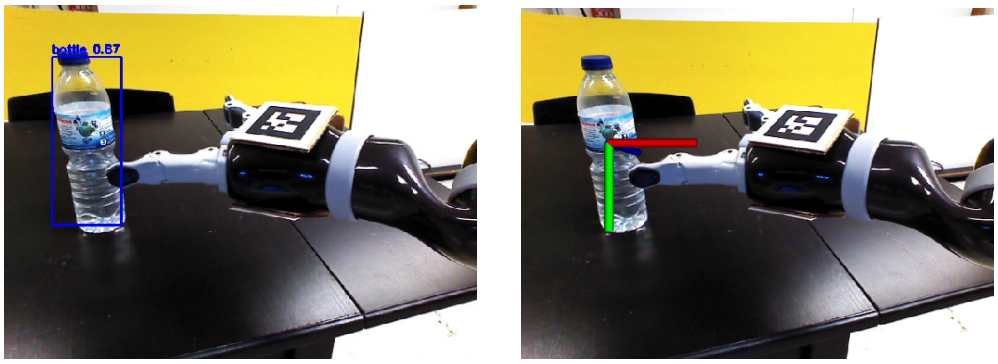


Figure 5.2: Target position estimation: **Left:** YOLO bounding box recognizing the bottle. **Right:** 3D position estimation based on the bounding box's central region depth information.

Figure 5.3: ALVAR markers used for end-effector pose estimation

Only one visible marker is needed for accurate 3D pose estimation, but to improve camera visibility of the wrist independently of its rotation, we placed three different markers (IDs 0-2) around the wrist, as seen in Fig. 5.3. A bundle file is created which stores the relative position of two markers (IDs 1 and 2) relative to the main marker (ID 0), as well as the size of the markers. When any marker is visible, the ALVAR package outputs $\hat{M}_m^c$ — the pose of the main wrist marker m in camera frame. However, for visual-servoing it is useful to locate the end-effector tool (or hand) which we want to bring closer to the target object. To do this, a fixed transform t_h^m is defined from the hand to the marker frame, based on measurements. A simple frame transformation gives the hand pose: $\hat{h}_h^c = \hat{M}_m^c t_h^m$.

5.4 Pregrasp Pose Selection

The purpose of this work was more on using visual-servoing to achieve a given grasp point, and less focus was put on selecting an appropriate grasp pose. However, since our system is designed to be modular and extensible, a simple grasp selection algorithm was developed to showcase the capability.

Our selection criteria is based on how close the target object is to the table, regarding height. The estimated height of the object is subtracted from the table's height — obtained as a parameter, but can also be obtained from an AR marker placed flat on the table's surface — and compared to a threshold (4cm in our tests). If the object's height is bigger than the threshold, the wrist is placed parallel to the table, as if to pick up a water bottle or coke can. Otherwise if the object is short, the robot's wrist is lifted in order to avoid colliding with the table. Fig 5.4 shows the two poses.

These serve as *pregrasp* poses. Since the pregrasp phase cannot benefit from visual feedback (the

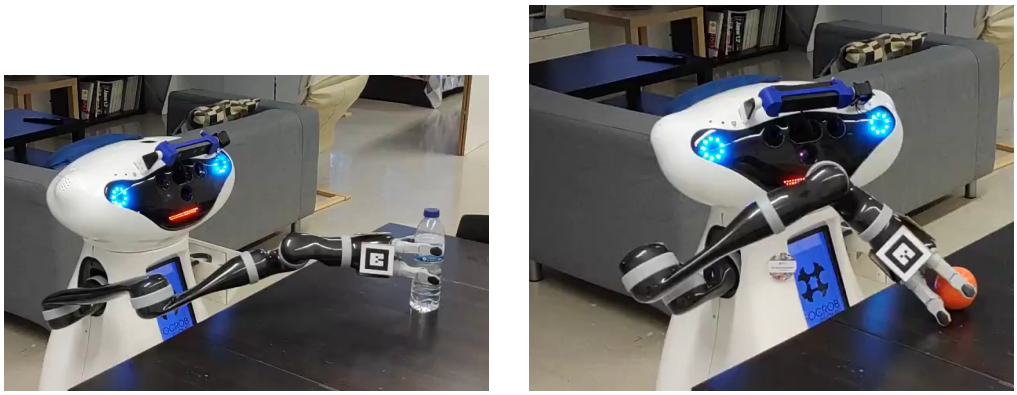


Figure 5.4: Grasps using the two developed pregrasp poses.

AR tags are still not visible to the camera), the arm is placed at a small distance from the object, close enough to make the AR tags visible to the camera. For the final approach (*grasp* phase) visual-servoing starts and controls only the linear velocity of the end-effector: its orientation stays the same as in the chosen pose.

5.5 Visual Servo Control

After the end-effector is placed in the pregrasp pose and the marker(s) are visible to the camera, visual servo control starts. The positioning error is calculated in camera-frame as the difference between the target and end-effector positions obtained from the methods above.

Since we wish to obtain the end-effector velocity in the arm's root frame (0), we need to transform the error to that frame. MBot's depth-camera is fixed to its head, and its position in head frame is given by T_c^{head}. The head can rotate left and right using a servo actuator. The head position relative to the body frame (`base_link`) is given by $\hat{T}_{head}^{base\_link}$, function of the servo's current angle. The arm is fixed to the robot's body according to transform $T_{base\_link}^0$. Multiplying these transforms yields the transformation matrix necessary to represent the camera-frame error in the root frame:

$$\hat{T}_c^0 = T_{base\_link}^0 \, \hat{T}_{head}^{base\_link} \, T_c^{head} \tag{5.1}$$

For proportional control it is also required to set $k > 0$, the gain factor. This value is configurable in our system, and we settle on $k = 12$ as it allows for a smooth error curve and adequate convergence time in our experiments.

Having obtained the required variables, we can now use the proportional control law given in (3.11) to perform position-based visual-servoing, obtaining the linear velocity required for the end-effector tool to approach the target object at every timestep:

$$u^0 = -k \, \hat{T}_c^0 (\hat{h}^c - \hat{g}^c) \tag{5.2}$$

Controlling the arm using this law naturally makes the error $(\hat{h}^c - \hat{g}^c)$ diminish through time. For the control loop to stop, we set a *stopping distance* parameter s: the acceptable error value at which

visual-servoing stops, so that the object is grasped by the gripper. s is also easily configurable, and we use $s = 0.01$ (1cm) in experiments.

Although Kinova's arm driver supports Cartesian velocity control of the end-effector (sending u^0 directly to the controller), this mode of operation has some flaws, e.g. not being able to activate the mode for certain arm positions. For this reason we decided to calculate the required joint velocities $\dot{q}$ and send those to the driver's joint velocity control mode, which works well. Doing this means solving the *inverse differential kinematics* problem described in section 3.1.3. The Jacobian matrix $J(q)$ is calculated at every timestep, based on the arm's kinematic structure and current joint angles q published by the arm driver. Singular value decomposition is performed to obtain the Jacobian pseudo-inverse $J^{\dagger}(q)$:

$$J = UDV^T, J^{\dagger} = VD^{\dagger}U^T \tag{5.3}$$

The joint velocities can now be obtained, replacing $J^{-1}(q)$ with $J^{\dagger}(q)$ in (3.7):

$$\dot{q} = J^{\dagger}(q)\ u^0 \tag{5.4}$$

We make use of the Eigen C++ library [40] to perform numerical operations efficiently.

5.6 Complete Pipeline Implementation

The developed visual-servoing control scheme is capable of making the end-effector approach the target object, but by itself does not constitute a grasping system. A complete pipeline was developed around it making use of the ROS framework, primarily the MoveIt! package. The pipeline is implemented as a state machine, which makes explaining each step more convenient. Below the pseudo-code for the state machine is presented, and each of its components is then described.

Function *GraspObject(object_type)*:

```
start_localizer(object_type)
sweep_head()
turn_head_to_object(object_tf)
pose ← select_pregrasp_pose(object_tf)
pregrasp(pose)
visual_servo()
close_gripper()
lift_and_lower_eef()
open_gripper()
go_to_pose('mbot_resting')
```

Algorithm 1: Object grasping pipeline

The system is started by running a ROS launch file, which includes system configuration parameters such as the visual-servoing stopping distance, or a flag to enable target tracking (more on that below). Afterwards, the grasping state machine can be triggered by calling a ROS service, giving it a single input: the type of object to grasp. This corresponds to the class name used in the YOLO classifier's

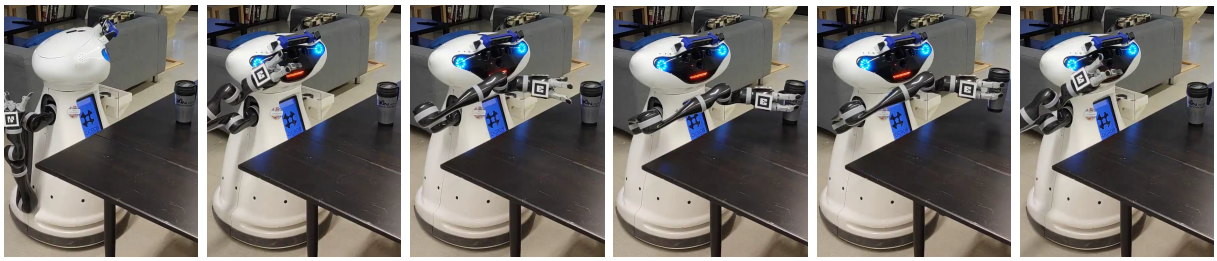


Figure 5.5: Steps of a grasping episode. **Left to right:** sweeping head, pregrasping, visual servo start, visual servo end and closing fingers, lifting object, moving back to resting pose after lowering object and opening fingers

output.

When the service is called, the object localizer node is started, which loads and runs the YOLO network (using the robot's GPU for acceleration) and the depth estimator for the given object type. This node publishes the object's position ($object\_tf$) as a ROS tf [41] transform.

The `sweep_head` procedure turns MBot's head left and right to build an octomap of the scene, allowing for a wide map of the obstacles to avoid in pregrasp motion planning. The head is then turned to point in the direction of the target object, using the broadcast transform $object\_tf$. An offset is added to turn the head slightly to the right of the object, so that the wrist marker stays visible in the camera frame.

`select_pregrasp_pose`($object\_tf$) performs the grasp pose selection algorithm described in section 5.4, taking into account the object's height in relation to the table. A motion plan to the selected pose is obtained through MoveIt!'s OMPL planner, using the RRT-Connect algorithm [36], and executed by the arm in a point-to-point trajectory with no feedback. The pregrasp pose puts the end-effector at some distance from the object (8cm, configurable).

With the end-effector now in camera frame, the visual-servoing procedure starts. As explained above, the end-effector approaches the target object until the stopping distance s is reached, after which the gripper's fingers are closed and the object is grasped. To check if a firm grasp has been achieved, the end-effector is lifted and then lowered using the `move_eef_relative(x, y, z)` function added to MBot Class for convenience. This function uses MoveIt!'s API to obtain the end-effector pose, applies the translation given by the arguments, plans the motion to the resulting pose and executes the movement. Finally the gripper is opened and the robot returns to the fixed `mbot_resting` pose, again using an MBot Class manipulation function `go_to_pose()`. Fig. 5.5 shows the pipeline's steps, and a video showing several grasps can be seen at `https://youtu.be/CZaLNTZ_ITU`.

The grasping launch file contains a flag argument `target_tracking` which determines whether the target object's position will be continuously tracked by the localizer. If this flag is false, it is assumed that the target object will remain stationary during the grasping process, and its position is only obtained at the start — the end-effector pose is still continuously estimated, allowing for visual-servoing. We decided to keep this flag disabled, as when the end-effector approaches the target object its fingers occlude the object's image, causing issues in target localization. This decision usually has no impact on grasping performance, as in the majority of cases the object remains stationary before being grasped.

Chapter 6

Results and Discussion

6.1 Experimental Setup

To test our system we use ERL's "Object Grasping and Manipulation" functionality benchmark [42]. Designed to be used in ERL Consumer competitions for domestic robots, this benchmark focuses solely on evaluating grasping and manipulation capabilities. Using this benchmark provides multiple advantages: it is a way to obtain objective results, allowing us to compare our method to other grasping pipelines, or evaluating differences between multiple versions of our system. Additionally, by using the same benchmark as the ERL, we can more easily prepare for and preview our performance in future competitions where SocRob participates. Finally, by using ERL's referee, scoring and benchmarking box (RSBB) [43], tests can be done in a semi-autonomous way and result logs with performance information are automatically saved. Implementation of this and other functionality benchmarks in RSBB was done by me, as part of a research grant coordinated by Dr. Meysam Basiri.

The RSBB is used to communicate with the robot by sending a preparation signal at first, and then sequential grasping goals consisting of the type of object to grasp and a target 2D position on the table where the object should be placed by the robot. To automatically score the benchmark, the RSBB also communicates with a motion capture (MoCap) system: retroreflective markers are attached to the objects (see Fig. 6.1) so that their pose can be detected by MoCap cameras. Two evaluation criteria are used: a) Was the object successfully grasped? b) How close to the target position did the robot place the object? A grasp is registered as successful if the object's height increased by more than 2cm (configurable in RSBB's settings). Closeness to the target pose is calculated based on the distance of the final object position to the given target coordinates. Because our focus is only on grasping and since there is no reference point for the table's origin position, we only consider the "successful grasp" scoring criteria.

Experiments were done in the ISRoboNet@Home Testbed in ISR [44], which features an apartment layout with real furniture and domestic objects, in addition to a MoCap system. Grasping was tested on a regular dining table, with 74cm height.

To use the functionality benchmark, we developed an integration between our grasping system and

Figure 6.1: MoCap markers placed on an object, for the benchmarking system to detect its pose

Figure 6.2: The objects used to test the grasping system

RSBB: after receiving a grasp goal, we read the object type and trigger our grasping state machine. After it finishes, we send an `end_execute` signal to the RSBB, which saves the benchmark's score and asks the operator to place the next object on the table.

A 25x20cm rectangular region of a dining table was used to test how the target position affects grasping performance. The region is divided into 9 positions (3x3, see Fig. 6.3). The object can be to the left, right or center of the robot, and it can be close, far or at middle distance. The region is relatively small because we assume the robot has navigated to a suitable position to grasp the object. Still, 25x20cm does not impose a high precision requirement for the navigation system, which can make use of the same target-localization system used for grasping and described in the previous chapter.

Five household items were chosen to test how the grasping system adapts to different objects: a water bottle, a tall mug, a large coffee cup, an espresso cup, and an orange (see Fig. 6.2). These were chosen not only because they are common items used in robotics competitions, but also to test both pregrasp poses chosen by our system: since the espresso cup and the orange are shorter objects, the system chooses the angled-wrist pregrasp pose when grasping them.

The robot attempted to grasp each of the 5 objects at every position, for a total of 45 (5x9) test grasps. The total percentage of successful grasps was measured, as well as the success rate per-object and per-position. While testing, we gathered data from two sources: the RSBB system and the robot itself. The RSBB data logs provide testing and performance data: For every grasp, the object name was saved along with its position on the table and whether it was successfully grasped. We also captured internal data from the robot to better analyze and debug the system. During each grasp we recorded the pose of the end-effector and target object, and during visual-servoing we recorded its error signal. The angles and velocities of each joint were also recorded.

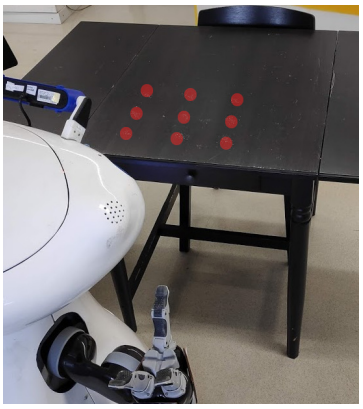


Figure 6.3: 3x3 grid of grasping zones

y(m) / x(m)	-0.125	0.00	0.125
0.20	**60%**	**60%**	**60%**
0.10	**100%**	**100%**	**100%**
0.00	**80%**	**80%**	**100%**

Table 6.1: Grasp performance by table position. Aggregate of all 5 objects.

Bottle	Tall mug	Large coffee	Espresso	Orange
88.8%	**88.8%**	**100%**	**66.6%**	**66.6%**

Table 6.2: Grasp performance by object

6.2 Results

This section describes the results obtained for the grasping benchmark, showing figures and internal data plots for some demonstrative grasps. All attempts were recorded on video, and a sample is available at `https://youtu.be/CZaLNTZ_ITU`.

Results are also shown for a comparison test between our version and one without visual feedback, assessing how they tolerate a measurement error of the arm.

6.2.1 Grasping Benchmark

The system achieved an overall grasping success of 82.2% — 37 successful grasps out of 45. Tables 6.1 and 6.2 show the grasp success per-position and per-object, respectively. Failures close to the robot (once for the mug and once for the bottle — Fig. 6.5 *f)*) happened due to the robot not being able to see the end-effector marker after pregrasping, failing to localize it and computing a velocity in an unwanted direction. When the orange and espresso cup (for which the angled-wrist pregrasp pose is chosen) were placed in the 3 positions furthest from the robot, the grasps always failed, due to the arm reaching a kinematic limit during visual-servoing. On two of those occasions the robot could still grip the object, though it couldn't be lifted (e.g. Fig. 6.5 *d)*). In one occasion the end-effector never reached the desired distance to the object — Fig. 6.5 *e)*.

6.2.2 Data Analysis

The error plots (Fig. 6.4) show the expected system behavior: A smooth motion is applied to the arm while pregrasping, and when visual-servoing starts the error follows an asymptotic function converging to 0, stopping at the desired distance of 1cm. The slight increase in the error value when switching the

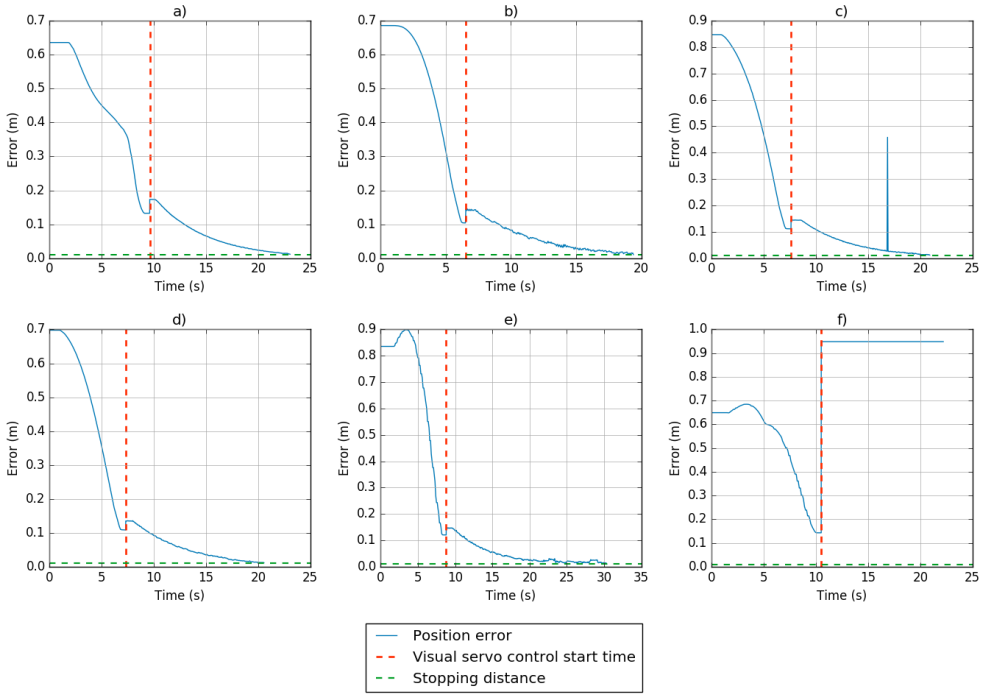


Figure 6.4: Plots of six different grasping episodes. The red dashed line is the time at which the pregrasp motion stops and visual servo control starts, continuing until the desired stopping distance is reached and the gripper closes. **a)** - **c)** correspond to successful grasps; **d)** and **e)** correspond to grasps on the table position furthest from the robot, causing a kinematic limit to be reached, but still gripping the object; **f)** corresponds to a failed grasp where the AR tag was never well-localized.

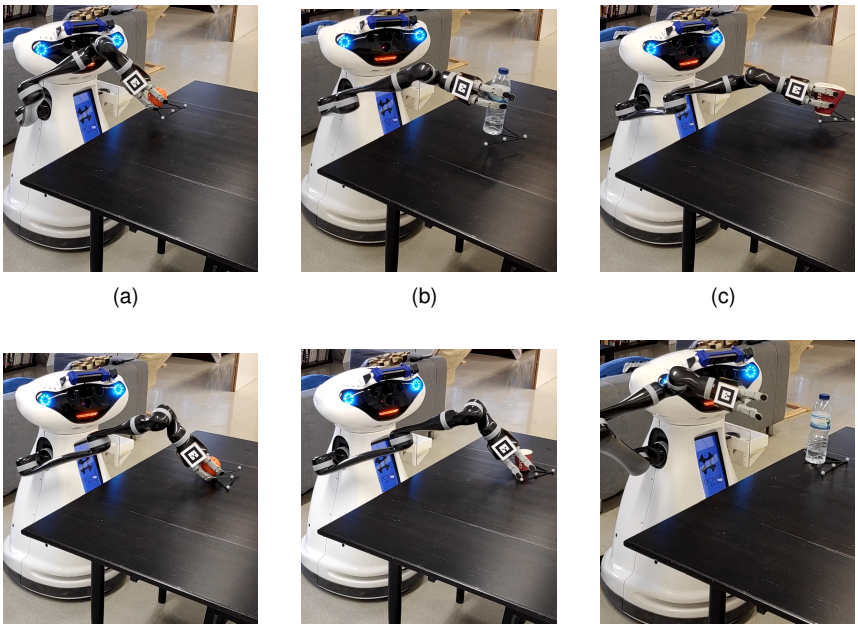


Figure 6.5: Pictures of the six grasping episodes referred in Fig. 6.4

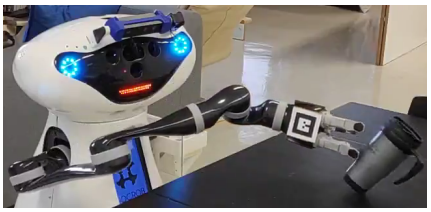
mode to VS happens because the point being considered is not the same: for pregrasping, the center of the end-effector link is considered, while visual-servoing considers a ”tool” link relative to the main AR tag. Additionally some of the error is caused by the camera estimation of the end-effector position being different than the kinematic estimation from the joint angles.

Plots **a)** - **c)** correspond to successful grasps. In **c)** there is a momentary spike in visual-servoing error, caused by a wrong localization of the end-effector tags by the ALVAR library, but quickly recovered. In plot **e)** a kinematic limit is reached before reaching the object, preventing the positional error from lowering, and therefore failing to grasp the object. Plot **f)** corresponds to a failure in localizing any wrist tag. The error stays at a high value, and the arm never approaches the object.

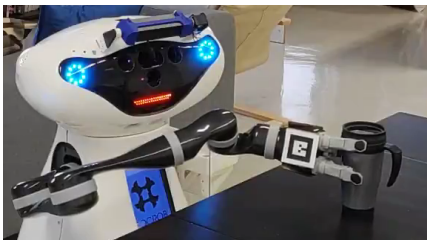
6.2.3 Error Tolerance Tests

To test our system's ability to handle calibration errors, we added an erroneous translation to the arm in relation to MBot's body, moving it 8cm to the right in the URDF measurements. We then tested two versions of the pipeline: a modified version which does not use visual feedback, making the final approach to the object only based on joint sensors and measurements, and our version which uses visual-servoing for the final approach. Three objects (bottle, tall mug and orange) were placed in the same table position for a fair comparison between the versions.

As expected, the version with no visual feedback places the hand on the left side of the object (since the arm's real position is 8cm to the left of the wrongly-assumed position). The grasps failed on the three objects (Fig. 6.6a shows one). When using our pipeline, while the pregrasp pose also places the end-effector to the left, visual-servoing then activates and starts correcting the error, guiding the arm to a successful grasp on all three objects (Fig. 6.6b).



(a) No visual feedback

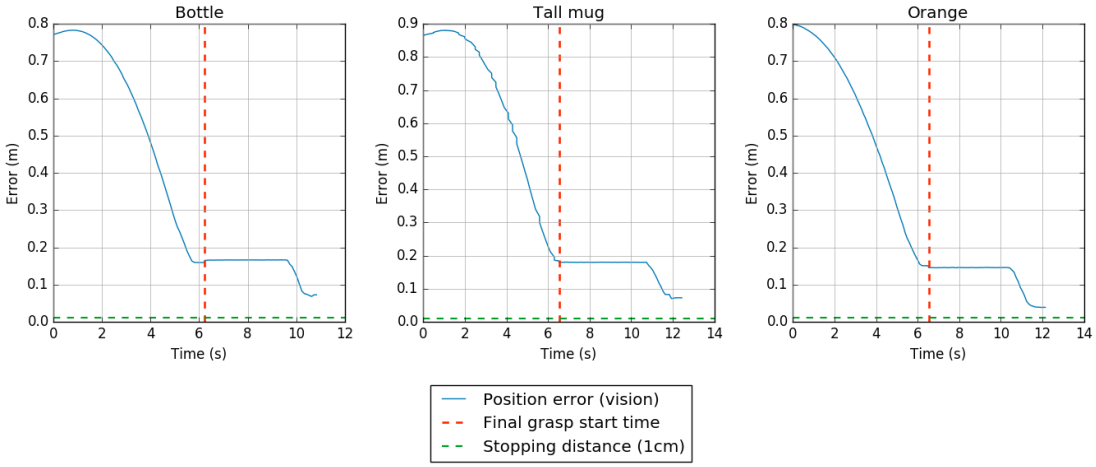


(b) Visual-servoing (ours)

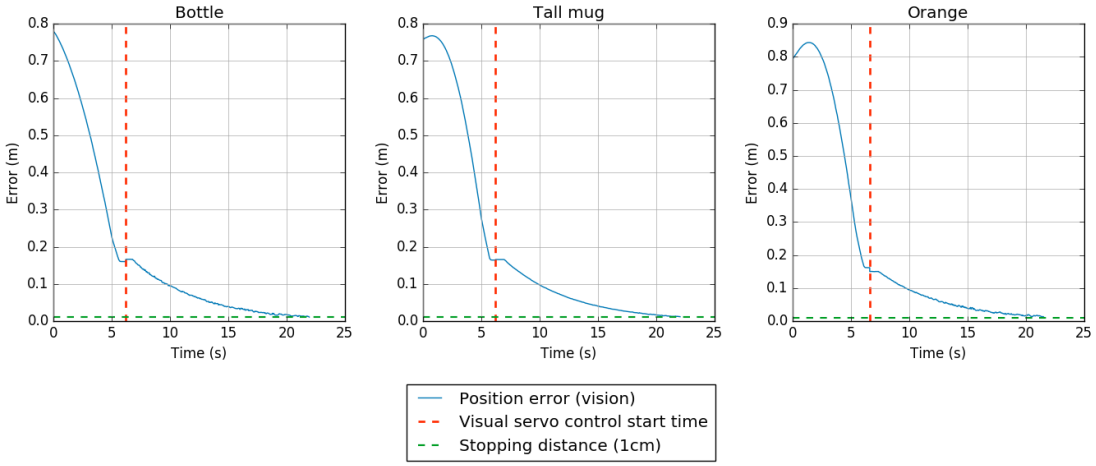
Figure 6.6: Comparison of grasping with and without visual feedback, with a wrong arm measurement (8cm to the right). Visual-servoing was able to correct the error.

We collected the same data for these tolerance tests as done above. In the modified no-feedback

version, the ALVAR tracker is still used to locate the AR wrist tags, but only to gather error data. Plots (Fig. 6.7) show that the no-feedback version takes about 4 seconds to calculate the plan of the final approach motion and then quickly executes it, but the positioning error stays large. The version using visual-servoing takes more time to make the approach, but the error converges to the desired 1cm value.



(a) No visual feedback



(b) Visual-servoing (ours)

Figure 6.7: Plots comparing the versions with and without visual feedback, when a wrong 8cm translation is added to the arm. Plots correspond to three objects placed at the same table position.

6.3 Discussion

Based on the results, the system shows consistency in achieving grasps on different objects. The bottle and mug failures both happened in positions close the the robot, where the initial pregrasp motion placed the wrist outside of the camera's field of view. In an integrated system where the robot can navigate to the table before triggering our grasping pipeline, this error could be avoided by simply moving away from the object when it is too close. The angled-hand pregrasp pose, while suitable for close positions, failed when objects were too far away. Navigation could also help here: by rotating the robot's base, the reach

can be extended.

The overall grasp success rate of 82.2% shows regular household objects can be consistently grasped in several positions, an important capability for the domestic tasks carried out by the SocRob team, and in robotics competitions.

Comparison tests with and without visual feedback show that our system can tolerate erroneous kinematic measurements, correcting the end-effector's position in camera frame. This validates the advantage of using visual-servoing in our system.

Chapter 7

Conclusions

7.1 Achievements

Our proposed system showed success in grasping several household items in different positions. Using visual-servoing makes it tolerant to calibration errors in the camera and the arm kinematics, allowing for a precise final grasp position. For the objects tested, the grasp poses achieved are firm enough to lift the objects.

Our target localization system, combining machine learning for object detection with depth data, proves to be useful for SocRob's purposes: many household items are detectable by pretrained models, and a specific detector can be trained to make the system able to grasp other objects. Using AR tags to obtain the end-effector pose is a pragmatic but effective way to obtain visual feedback of the arm's motion.

By developing a grasping pipeline around the visual-servoing approach, we make it possible to trigger a grasp by simply setting the object type, and the robot will move the head to create the scene's octomap, grasp the object avoiding collisions, and return the arm to its resting position. This enables the easy use of the pipeline as a part of more involved domestic and assistive tasks. Additionally, our software and hardware modifications allow other MBot users to interface with the Kinova arm for different purposes.

7.2 System Limitations

The main limitation of our system has to do with grasp pose selection. Although we have made the system able to accept any grasp selection algorithm, we developed a simple one just to exemplify the possibility. As it is, the arm is always placed at one of two possible pregrasp poses, one more suitable for cylindrical objects (bottles, cups, drink cans, etc.) and another for shorter objects, closer to the table surface. This causes failures (such as the ones seen in the previous chapter) that could be avoided by selecting a better grasp pose.

Another related limitation is the fact that visual-servoing controls only *translation* and not *rotation*. The system plans a pregrasp motion in which the end-effector is already at the desired grasp angle, and

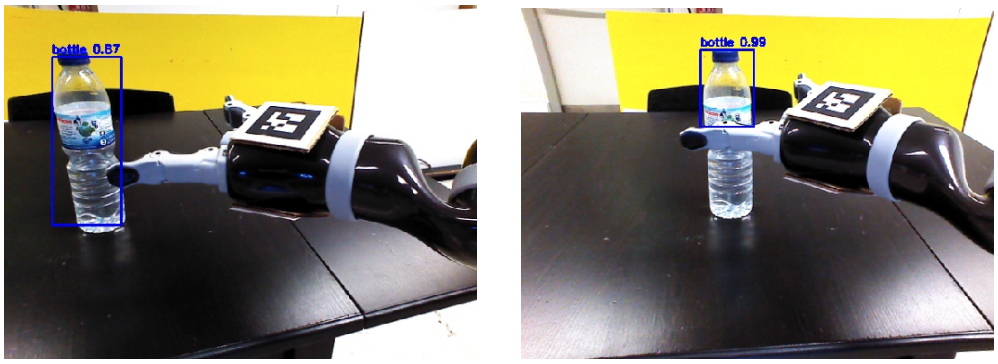


Figure 7.1: Target localization failure caused by occlusion.

visual-servoing only applies linear (not angular) velocity to the end-effector.

Additionally, our system does not track the target object's pose. Although there is a flag to enable it, target tracking causes localization failures due to occlusion by the end-effector: as Fig. 7.1 shows, when the arm's fingers come close to the object they hide it, causing errors not only in image detection, but also in depth perception. With this flag disabled, visual feedback is still obtained by tracking the end-effector tags, but the system assumes the target object remains stationary, which is usually the case, including in our tests.

7.3 Future Work

This work could be extended by adding a more robust grasp pose selection algorithm, which could take into account depth-camera data, and potentially images of the object from different angles — provided by applying movement to MBot's mobile base — stitched together to perform 3D object reconstruction, using existing models of the objects to grasp.

The grasping pipeline could be extended by adding navigation at the start: after identifying the target object's position, the robot can move to an advantageous position, improving the arm's reach and grasp feasibility.

Before visual-servoing starts, the final grasp position could be estimated to calculate its feasibility. Additionally, a *manipulability measure* could be calculated during visual-servoing, analyzing the Jacobian matrix to detect if the arm is close to a kinematic singularity.

Bibliography

[1] M. Consortium. Deliverable d2.1.1 (update) - monarch robots hardware, 2014. URL http://users.isr.ist.utl.pt/~jseq/MOnarCH/Deliverables/D2.2.1_update.pdf.

[2] B. Singh, N. Sellappan, and P. Kumaradhas. Evolution of industrial robots and their applications. 2013.

[3] J. McCarthy, L. Earnest, D. R. Reddy, and P. J. Vicens. A computer with hands, eyes, and ears. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 329–338, 1968.

[4] S. Hutchinson, G. D. Hager, and P. I. Corke. A tutorial on visual servo control. *IEEE Transactions on Robotics and Automation*, 12(5):651–670, 1996. ISSN 1042296X. doi: 10.1109/70.538972.

[5] Y. Shirai and H. Inoue. Guiding a robot by visual feedback in assembling tasks. *Pattern Recognition*, 5(2):99 – 108, 1973. ISSN 0031-3203. doi: 10.1016/0031-3203(73)90015-0. URL http://www.sciencedirect.com/science/article/pii/0031320373900150.

[6] J. Hill. Real time control of a robot with a mobile camera. 1979.

[7] F. Chaumette and S. Hutchinson. Visual servo control. I. Basic approaches. *IEEE Robotics and Automation Magazine*, 13(4):82–90, 2006. ISSN 10709932. doi: 10.1109/MRA.2006.250573.

[8] P. I. Corke. *Visual Control of Robot Manipulators — A Review*, pages 1–31. doi: 10.1142/9789814503709_0001. URL https://www.worldscientific.com/doi/abs/10.1142/9789814503709_0001.

[9] C. Collewet and E. Marchand. Photometric visual servoing. *Robotics, IEEE Transactions on*, 27: 828 – 834, 09 2011. doi: 10.1109/TRO.2011.2112593.

[10] H. Hashimoto, T. Kubota, Wai-chau Lo, and F. Harashima. A control scheme of visual servo control of robotic manipulators using artificial neural network. In *Proceedings. ICCON IEEE International Conference on Control and Applications*, pages 68–69, 1989.

[11] Q. Bateux, E. Marchand, J. Leitner, F. Chaumette, and P. Corke. Training deep neural networks for visual servoing. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3307–3314, 2018. doi: 10.1109/ICRA.2018.8461068.

[12] M. Zhu, K. Derpanis, Y. Yang, S. Brahmbhatt, M. Zhang, C. J. Phillips, M. Lecce, and K. Daniilidis. Single image 3d object detection and pose estimation for grasping. *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3936–3943, 2014.

[13] I. Lenz, H. Lee, and A. Saxena. Deep learning for detecting robotic grasps. *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, 2013. doi: 10.15607/rss.2013.ix.012.

[14] S. Kumra and C. Kanan. Robotic grasp detection using deep convolutional neural networks. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 769–776, 2017. doi: 10.1109/IROS.2017.8202237.

[15] D. Morrison, P. Corke, and J. Leitner. Closing the loop for robotic grasping: A real-time, generative grasp synthesis approach. In *Robotics: Science and Systems XIV*, pages 1–10. Robotics Science and Systems Foundation, 2018. doi: 10.15607/RSS.2018.XIV.021. URL `https://eprints.qut.edu.au/121236/`.

[16] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17:1–40, 2016. ISSN 15337928.

[17] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. volume 87 of *Proceedings of Machine Learning Research*, pages 651–673. PMLR, 29–31 Oct 2018. URL `http://proceedings.mlr.press/v87/kalashnikov18a.html`.

[18] K. Deguchi. A direct interpretation of dynamic images with camera and object motions for vision guided robot control. *International Journal of Computer Vision*, 37(1):7–20, June 2000. ISSN 0920-5691. doi: 10.1023/A:1008151528479. URL `https://doi.org/10.1023/A:1008151528479`.

[19] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 3, pages 2619–2624 Vol.3, 2004.

[20] T. Haarnoja, A. Zhou, S. Ha, J. Tan, G. Tucker, and S. Levine. Learning to walk via deep reinforcement learning. *ArXiv*, abs/1812.11103, 2019.

[21] N. Heess, T. Dhruva, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. Eslami, M. A. Riedmiller, and D. Silver. Emergence of locomotion behaviours in rich environments. *ArXiv*, abs/1707.02286, 2017.

[22] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 1st edition, 2008. ISBN 1846286417.

[23] S. Schaal. Jacobian methods for inverse kinematics and planning, 2014. URL `https://homes.cs.washington.edu/~todorov/courses/cseP590/06_JacobianMethods.pdf`.

[24] V. Lippiello, B. Siciliano, and L. Villani. Eye-in-hand/eye-to-hand multi-camera visual servoing. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 5354–5359, 2005. doi: 10.1109/CDC.2005.1583013.

[25] A. C. Sanderson and L. E. Weiss. *Adaptive Visual Servo Control of Robots*, pages 107–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. ISBN 978-3-662-09771-7. doi: 10.1007/978-3-662-09771-7_7. URL https://doi.org/10.1007/978-3-662-09771-7_7.

[26] S. Haykin. *Neural Networks and Learning Machines*. Pearson Education, third edition, 2008. ISBN 9780133002553. URL https://books.google.pt/books?id=faouAAAAQBAJ.

[27] S. University. Cs231n convolutional neural networks for visual recognition, 2020. URL https://cs231n.github.io/convolutional-networks.

[28] Y. Lecun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, pages 21–28. Morgan Kaufmann, 1988.

[29] MathWorks. Deep learning - convolutional neural network, 2020. URL https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html.

[30] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[31] O. Robotics. Urdf package, 2020. URL http://wiki.ros.org/urdf.

[32] O. Robotics. xacro package, 2020. URL http://wiki.ros.org/xacro.

[33] M. project. Moveit!, 2020. URL https://moveit.ros.org.

[34] P. Beeson and B. Ames. Trac-ik: An open-source library for improved solving of generic inverse kinematics. In *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, pages 928–935, 2015. doi: 10.1109/HUMANOIDS.2015.7363472.

[35] I. A. Şucan, M. Moll, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. doi: 10.1109/MRA.2012.2205651. https://ompl.kavrakilab.org.

[36] J. J. Kuffner and S. M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 2, pages 995–1001 vol.2, 2000.

[37] T.-Y. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. *ArXiv*, abs/1405.0312, 2014.

[38] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *ArXiv*, abs/1804.02767, 2018.

[39] V. T. R. C. of Finland Ltd. Alvar sdk, 2020. URL http://virtual.vtt.fi/virtual/proj2/multimedia/alvar/.

[40] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2020.

[41] O. Robotics. Ros tf package, 2020. URL http://wiki.ros.org/tf2.

[42] M. Basiri, E. Piazza, M. Matteucci, and P. Lima. Benchmarking functionalities of domestic service robots through scientific competitions. *KI - Künstliche Intelligenz*, 33, 09 2019. doi: 10.1007/s13218-019-00619-9.

[43] R. . E. community. Referee, scoring and benchmarking box, 2020. URL https://github.com/rockin-robot-challenge/rsbb.

[44] ISR. Isrobonet@home testbed, 2014. URL https://welcome.isr.tecnico.ulisboa.pt/isrobonet/.

Appendix A

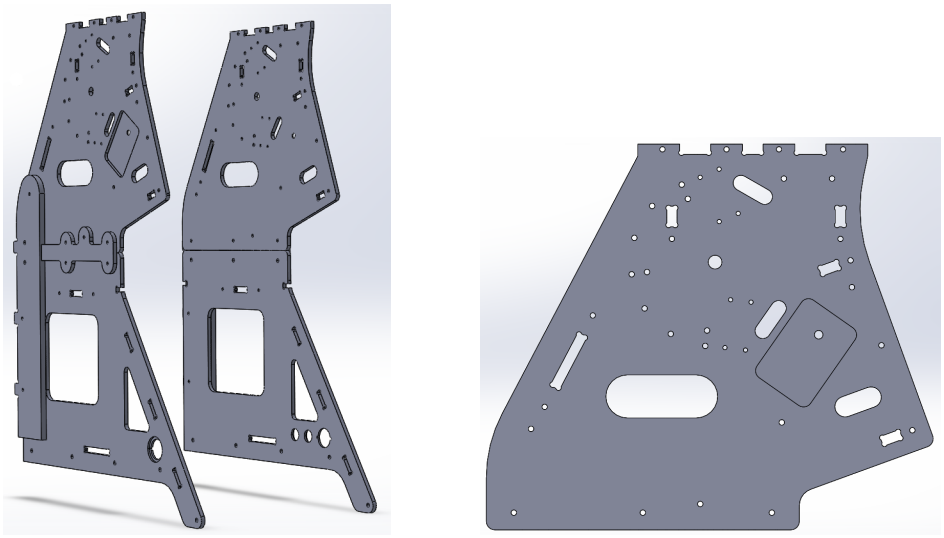
New MBot parts

This appendix includes pictures and descriptions of the MBot parts designed to support the newly-installed Kinova Gen2 arm. The parts were designed in SolidWorks and machined by IDMind (an IST spinoff company), with the exception of the pocket for the boost converter, which was 3D printed in-house at ISR.

A.1 Side-plates

A hole was added to the right side-plate, for the arm to pass through. An outer plate was designed to serve as a second layer, adding support. The outer plate connects the inner plate with the base, a task that was previously done by a smaller piece, visible in Fig. A.1 (a).

Note: In the figures below, the side-plates are positioned as if MBot was facing the viewer, slightly turned to the viewer's right side. This means the right side-plate is on the figure's left side, and vice-versa.



(a) (b)

Figure A.1: Original side-plates. **a)** MBot's body. **b)** Detail of right side-plate.

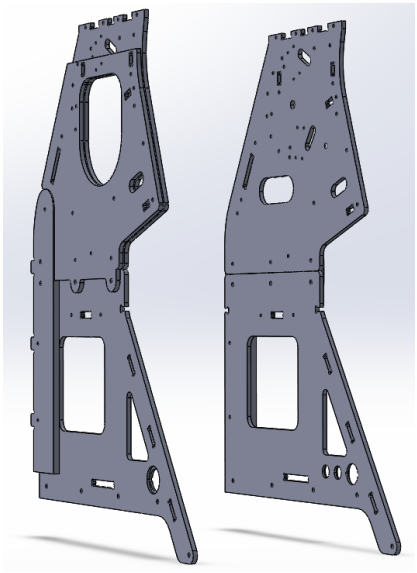


Figure A.2: Our design: MBot body with reinforced right side-plates to support the arm, and holes for it to pass through.

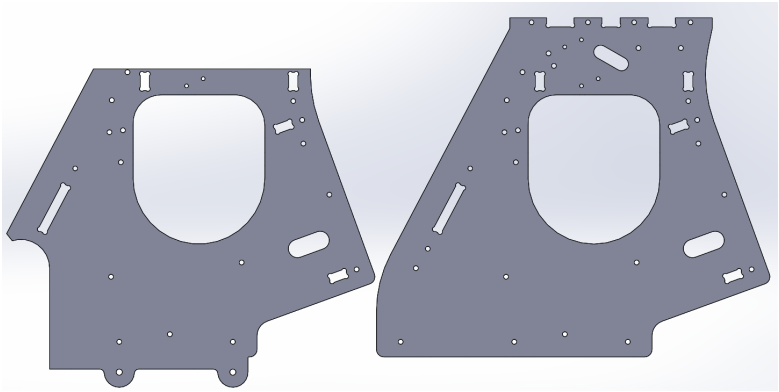


Figure A.3: Our design. **Left:** Outer layer. **Right:** Inner layer.

A.2 Mounting piece

A piece was designed to fix the arm base to MBot's left side-plate. The screw holes were placed asymmetrically to avoid going through important places.

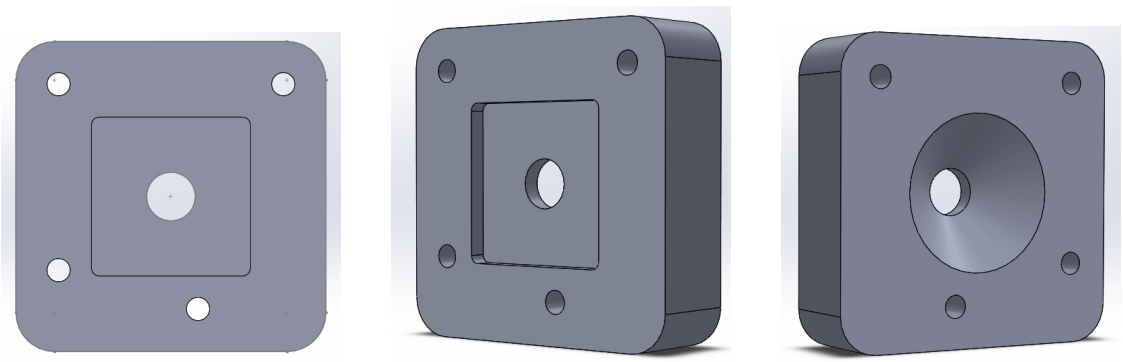
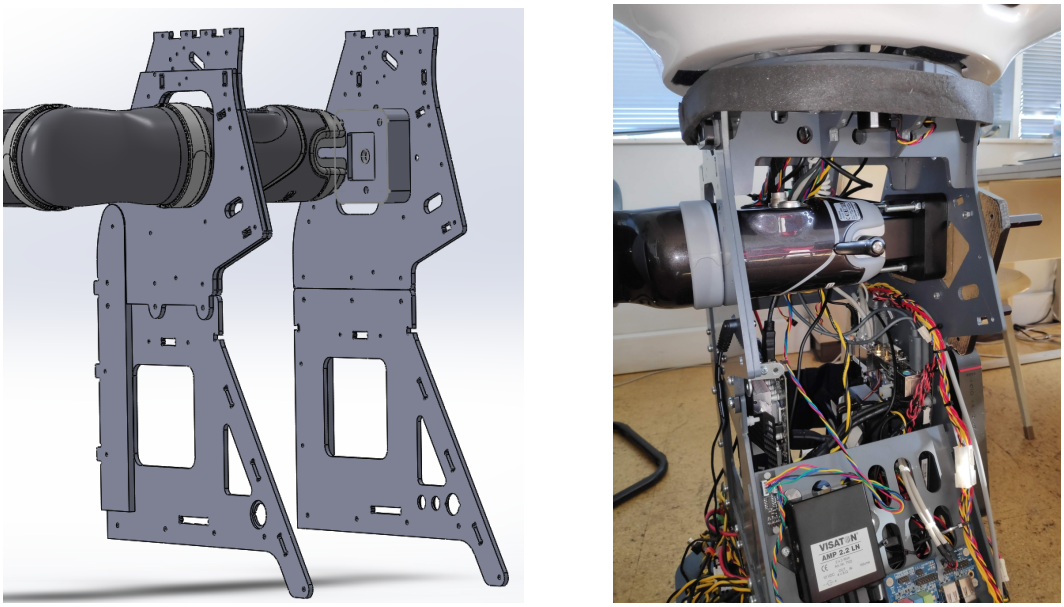


Figure A.4: Mounting piece to fix the arm to the left side-plate. Front, side, and back views.

A.3 Arm assembly

The parts were designed inside a SolidWorks assembly in which we included a model of the Kinova Gen2 arm. This way, measurements can be refined and the final result can be inspected. A part of the arm is not visible in the model: a metal brick which attaches to the arm's base and to the mounting piece. This part can be seen in the real assembly — Fig. A.5 (b).



(a) (b)

Figure A.5: Assembly of the arm inside MBot's body. **a)** SolidWorks assembly. **b)** Real assembly.

A.4 Boost converter and 3D-printed pocket

A PULS CD5.243 boost converter (Fig. A.6) was added to the robot, to convert from 12V DC power from one of the batteries to 24V DC power required by the Kinova Gen2 arm. The converter has a power limit of 96W, which matches the power consumption of the arm (25W average, 100W peak).

We designed a pocket to hold the converter (Fig. A.7) and installed it in the robot.

Figure A.6: PULS CD5.243 boost converter.

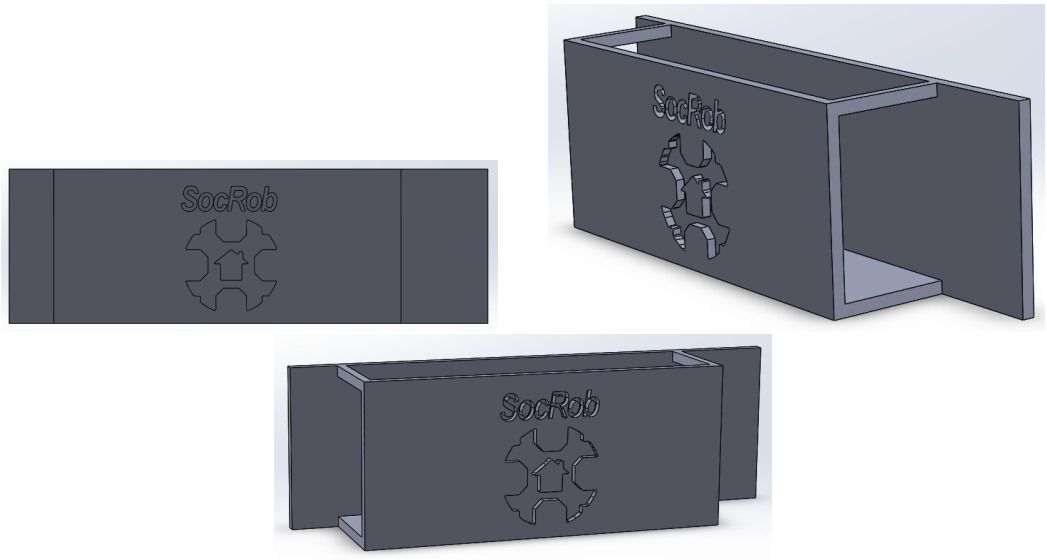


Figure A.7: SolidWorks model of the pocket holder.

Figure A.8: Converter inside MBot, held by the 3D-printed pocket.