FACULTY 1: PHYSICS AND ELECTRICAL ENGINEERING
INSTITUTE FOR ARTIFICIAL INTELLIGENCE

MASTER THESIS (INFORMATION AND AUTOMATION ENGINEERING)

PARAMETERIZATION OF A CONSTRAINT-BASED AND OPTIMIZATION-BASED ROBOTIC MOTION CONTROLLER FOR OBJECTS GRASPING IN TABLE-TOP SCENARIOS

WRITTEN BY
MINERVA GABRIELA VARGAS GLEASON (3020449)
ON SEPTEMBER 21, 2017

1ST SUPERVISOR: DR. ING. DANIJELA RISTIC-DURRANT
2ND SUPERVISOR: PROF. MICHAEL BEETZ, PHD
ADVISORS: GEORG BARTELS, ALEXIS MALDONADO

*EXZELLENT.

Contents

List of Figures

CHAPTER 1

Introduction

Motion planning and trajectory generation is a central part of robotics. Being able to parametrize the movements a robot executes is crucial for the flexibility of the system and an important step in the robot-human interaction field.

The number and applications of robots in our society is growing day by day, we can find them in production lines, at hospitals, as guides in museums and even at home, just to mention some examples. The capabilities and autonomy required by each robot widely varies depending on its application. One often addressed problem in robotics is object manipulation.

The aim of this work is to develop the *motion generation component* of a *grasping system* based on a whole-body robot motion controller. The system must be able to obtain a trajectory that a robot has to execute in order to grasp a specified object. This implies the system detects the object and generates a trajectory for a mechanical system with multiple degrees of freedom (DOFs) that successfully reaches a position and orientation under given initial conditions.

The developed system will be implemented and tested on the Boxy robot (figure 1.2), a dual-armed robot from the Institute for Artificial Intelligence[1] (IAI). The trajectory generation system currently used by Boxy is Giskard (section 2.5.3), an optimization-based controller devel-

[1] http://ai.uni-bremen.de/research/robots/boxy

oped at the IAI.

The main contributions of this work are implementing a collision detection system and limiting the acceleration of the robot while planning, as well as the ability of deciding on its own how an object should be grasped.

1.1 Motivation

In the 1960's the robots were introduced in the industry. They were mainly used at the automotive industry for pick and place tasks (Wallén, 2008). Since then, robots have been replacing humans in monotonous, hard and dangerous tasks.

Most industrial robots used in assembly lines have no sensors that gives them information about their environment, they work by moving from one predefined position to the next one, executing controlled trajectories. Robots working without external information have to be kept in a controlled environment, where the position of all objects is known beforehand and they only have to execute a repetitive task.

Nowadays, service robots are envisioned to work in a wide range of situations where they dynamically interact with their environment, where the position of objects and obstacles is not known beforehand and can continuously change. Working under these conditions requires some degree of autonomy from the system.

One of the central questions this thesis address is how can one parametrize the trajectory generation for a robot in an object grasping scenario, so that the robot is able to decide how it's movement should be under given conditions and successfully execute a grasping movement.

An important point of this work is to understand the behavior of an optimization-based and constraint-based robotic motion controller, to find out which constraints are relevant for the trajectory generation and how each one of these constraints affects the controller's output.

Object grasping and manipulation constitutes a big challenge in the robotics field. In order to successfully grasp an object, the robot must:

- Identify the object and its position with respect to the robot. This is done by a perception system which analyzes the information provided from sensors, normally cameras.
- Decide how to grasp the object. Usually, an object can be grasped in several ways. Taking a cup for example, it can be grabbed from the handle, from the cup body or from the top edge (figure 1.1). The system should be able to decide which one of this **grasping poses** is better depending on the situation.
- Generate trajectories that move the end effector (EEF) from its current location to the selected grasping pose. This includes trajectory generation and motion control.
- Evaluate if the obtained trajectories will generate a collision with the robot itself or the environment. Discard the trajectories where collisions were detected.
- Send the best trajectory to a PI controller that will send the corresponding commands to the robot to execute it.

Being able to grasp an object is an essential ability in robot manipulation. Having a system that autonomously locates and grasp an object using *On-line Trajectory Generation (OTG)* gives great flexibility to the robot. OTG grants the capability of modifying the trajectory during execution, it implies recalculating the trajectory on every control cycle . By doing this, the system

is able to react instantaneously to unexpected events, such as a change in the goal position.

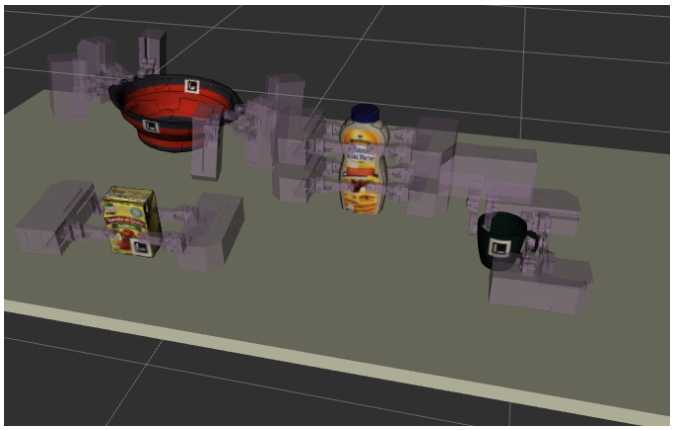


Figure 1.1: Example of objects with predefined grasping poses

The proposed solution to this grasping problem includes generating a data base of objects with predefined grasping poses (GP) (figure 1.1) and a perception system that detects the stored objects and shows the possible GP and it's position with respect to the robot.

The robot will have to decide which GP is better for grasping a specified object given a initial configuration. Then, an optimization controller (section 3.3) will generate several possible trajectories, evaluate them and send the best one to the Boxy robot (figure 1.2).

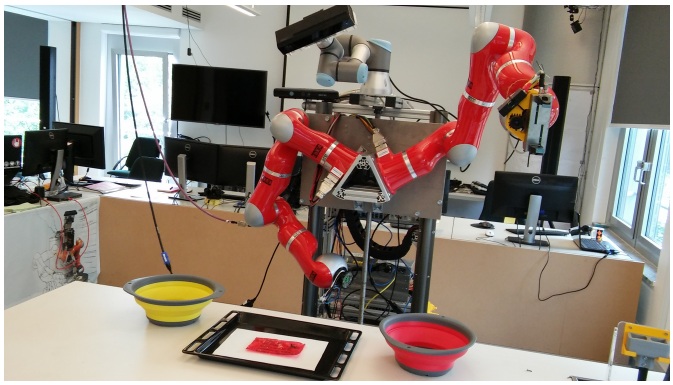


Figure 1.2: The Boxy robot

As result, a system with a parametrized motion controller that generates a smooth trajectory for Boxy by limiting the velocity and acceleration of each joint is expected. The system should be

able to decide which grasping pose (GP) of a selected object can be better grasped given the current system conditions.

1.2 Contributions

The main contribution of this work is the development of a projection system for trajectory planning (section 3.1) with a whole body motion controller for the robot. This system uses a naive kinematic simulator (section 2.5.5) of the robot to generate several possible trajectories. Each one of the obtained trajectories is evaluated and scored and one is selected to be executed by the real robot. Some of the evaluation criteria are trajectory smoothness and length, manipulability of the robot after the goal position is reached and collisions during the trajectory simulation.

The collision detection was developed using Moveit! Planning scene[2] (section 2.5.6). The system is not able to consider collision avoidance while generating a trajectory, but the collision detection system discards all trajectories that would generate a collision with the robot itself or with the environment (section 3.5).

During the development of this work, an object database was created(section 3.2.1) with 3D models of several objects and predefined grasping poses for each object. The object models were created using a 3D scanner. The gripper and furniture models were done in a CAD program.

In order to detect the objects located in front of the robot, I developed a perception system that uses fiducial markers as side product (section 3.2).

[2]http://moveit.ros.org/

The robot moves by receiving a sequence of velocity commands for the joints, so it is not possible to send a trajectory at once, it has to be sent step by step. A simple PI controller was created to send the obtained trajectory to the robot (section 3.6).

1.3 Structure of the thesis

This chapter has given an explanation of the aim of this work and the motivation behind it, as well as the outcomes of it. The second chapter explains the theory behind the algorithms of this thesis and some useful concepts. It starts by describing some basic concepts for trajectory planning in robotics, such as kinematic models and collision environment. Afterwards it explains some motion planning algorithms, including the one used in this work. In the end, it gives a short description of the software used by the system.

Chapter three depicts all the elements of the system created in this work. First, it shows the system architecture and starts describing them. At first, the perception system is explained, afterwards the structure of the database where the object's models and descriptions are stored. The third section explains the motion controller, the main component of this work. Then it gives a brief explanation of the trajectory evaluation and collision checking and afterward it describes the PI controller used to send the evaluated trajectories to the Boxy robot.

The fourth chapter displays the results of this works and describes the evaluation of the generated trajectories. It starts by describing the two types of experiments made: simulated and on the real robot, and the environment for each of them. Thereafter, it describes the behavior of the controller during one of the simulated trajectories. Then it shows tables with the evaluation of the generated trajectories. In the end, the results of the trajectories executed on the real robot

are shown.

The fifth and last chapter discusses the results shown in chapter four. It also provides a short summary about the contributions of this work and proposes further improvements. The outcomes and applications of this system are also stated.

For the ones interested in using this work, the appendix provides links to the Github repositories with all the code generated and a short explanation on how to execute it.

CHAPTER 2

Background & Related Work

This chapter presents the current state of the art of some relevant topics for mobile robotics that the reader needs to fully understand the content of this thesis. The first section will refer to the kinematic model of a robot. Afterwards, the general concepts of trajectory planning will be explained. In the end, a brief explanation of the software used in this thesis is given.

2.1 Kinematic Modelling of a Robot

According to Siliciano, Khatib (2007), *robot kinematics* refers to the motion of the elements in a robot without considering the forces and torques that generated this movement. To describe the movement of a robot, we need a kinematic model of its structure. The position and orientation of a body in space is known as *pose*.

A robot can be modeled as a kinematic chain, which consists of a system of rigid bodies connected by joints, a *kinematic joint* is a connection between two bodies that constrains their relative motion. In robotics, these bodies are usually referred to as *links*. The kinematic description of a robot normally uses some simplifications: the links that form the robot are assumed to be rigid, and each link is ideally connected with the next one, with no gap in between.

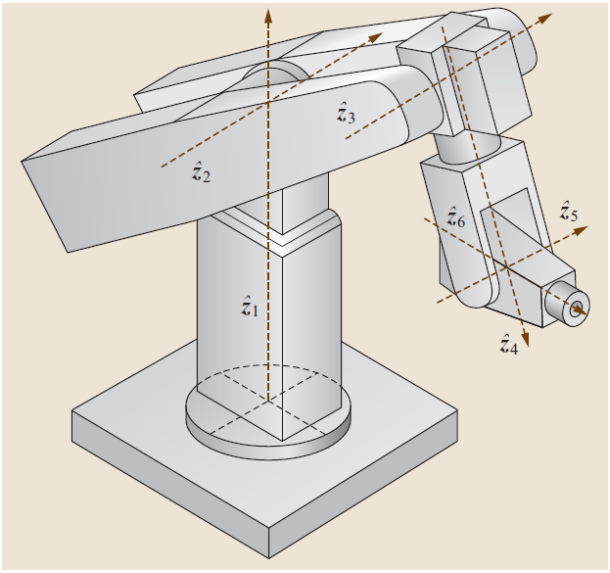


Figure 2.1: Example of a 6 DOF manipulator. (Siliciano, Khatib, 2007, chap. 1, page 24)

Figure 2.1 shows an example of a serial chain manipulator, where a reference frame is attached to each link. Following the Denavit-Hartenberg convention, the Z axis is always aligned with the joint axis (Craig, 2005).

The study of robot kinematics involve how the location of the reference frames change as the mechanism moves. The main goal is to compute the position and orientation of the manipulator's end-effector with respect to the base as a function of the joints values.

For an open-loop robotic mechanism, the general structure is represented by a kinematic tree, which consists of the concatenation of links and joints with a reference frame in each joint (Siliciano, Khatib, 2007). This representation is used to obtain a mathematical model of the system.

When programming a robot, a suitable representation of its kinematic model is required. The Unified Robot Description Format (URDF) is a standard XML representation of the robot model in the ROS community[1], which includes the robot kinematics, dynamics, and sensors.

[1] http://wiki.ros.org/urdf

2.2 Collision Environment

Workspace

The workspace of a robot is defined as the total volume swept out by the end-effector as the manipulator executes all possible motions (Siliciano, Khatib, 2007, chap. 1). In simple terms, the workspace is the space that can be reached by the robot.

Working environment

The working environment consists of all the external factors surrounding and affecting the robot. In other words, it is the space where the robot is working, including all the objects in there. According to Dmitry Berenson, Kuffner (2011), the robot is considered to be operating in a "world" that it cannot leave; actions can affect the environment and, thus, change it for the robot.

The environment can be represented in a graphical way with meshes that describe the geometry of the objects and of the robot. The position of the meshes in the model environment has to match the position of the real objects. Motion planning algorithms are then used to calculate trajectories that the robot can execute without colliding with the environment or itself.

Sensors

A robot can have two types of sensors (Russell, 1990, chap. 1):

- **Proprioceptive sensors:** Get information about the internal state of the robot, such as the angular position of each joint or the temperature of the links. These sensors are used for self maintenance and control of internal status. Examples of proprioceptive sensors are:

shaft encoders, inertial navigation systems and force sensors.

- **Exteroceptive sensors:** Obtain information from the robot's environment, such as the distance to an object relative to a frame of reference of the robot. Many exteroceptive sensors are sensors that can be used to calculate distances. These sensors can be further categorized in contact, range, and vision sensors.

The robot takes the information from proprioceptive sensors to calculate the position and orientation of each link and determine its current configuration, and the information from exteroceptive sensors to calculate its position relative to the environment and the distance to surrounding objects. Combining both, the motion planning algorithms can iterate until a collision-free path between the initial and desired position is found.

2.3 Trajectory Planning

According to Han et al. (2011, chap. 1, page 6) *"A trajectory is the path that a moving object follows through space as a function of time"*. A trajectory can be described as a time-stamped series of location points.

To define a robot's trajectory, we must define the trajectory that each link will execute to bring the robot to a desired pose. Once the robot's kinematic model and environment are defined, we use a motion planning algorithm to solve the path planning problem and obtain a collision-free trajectory for the robot. Planning involves determining the path and the velocity function for a robot.

2.3.1 Euclidean Space in Cartesian Coordinates

In geometry, we can describe the position of any object in space using Cartesian coordinates (X, Y, and Z). For a 3-dimensional space (Euclidean space), any object can move and rotate along 3 axes, this means that the object has 6 degrees od freedom (DOF), so 6 values are required to define the pose of an object with respect to a reference point. In other words, for a three-dimensional Euclidean space, we can describe the position and orientation of any object using the Cartesian coordinates and three rotation angles.

2.3.2 Configuration Space (C-space)

A complete description of the robot's geometry and of the workspace is needed to solve the path planning problem. A *configuration* $\boldsymbol{q}$ specifies the location of every point of the robot's geometry.

The *configuration space*, where $\boldsymbol{q} \in \boldsymbol{C}$, is the space of all possible configurations (Siliciano, Khatib, 2007). It represents all possible transformations that can be applied to the robot given its kinematics. The C-space gives an abstract way of solving the planning, the main advantage of this representation is that a robot can be mapped into the C-space as a single point, where the number of DOF of the robot is the dimension of the C-space. This is also the minimum number of parameters required to describe a configuration, so motion planning for the robot is equivalent to motion planning for the C-space.

During trajectory planning, one must consider several constraints involving the robot's pose. Since the allowed configurations of the robot are not known beforehand, the planning algorithm must find these valid configurations while planning. Finding these configurations is normally

done through sampling, this process can become inefficient for complex environments.

2.4 Motion Planning Algorithms

One of the fundamental problems in robotics is to plan motions for complex bodies from an initial pose to a goal. As explained by Siliciano, Khatib (2007), the general path planning problem is computing a continuous free path for the robot between $\boldsymbol{q}_1$ and $\boldsymbol{q}_G$, given:

1. The robot's workspace $\boldsymbol{W}$
2. An obstacle region $\boldsymbol{O} \subset \boldsymbol{W}$
3. A robot defined as a collection of m links: $A_1, A_2, ..., A_m$
4. The C-space $\boldsymbol{C}$ with defined $\boldsymbol{C}_{obs}$ and $\boldsymbol{C}_{free}$
5. An initial configuration $\boldsymbol{q}_1 \in \boldsymbol{C}_{free}$
6. A goal configuration $\boldsymbol{q}_G \in \boldsymbol{C}_{free}$

Where $\boldsymbol{C}_{obs}$ is the *C-space obstacle region* and $\boldsymbol{C}_{free}$ is the set of configurations that avoid collisions, called *free space*. We must compute a continuous free path for the robot between $\boldsymbol{q}_1$ and $\boldsymbol{q}_G$.

The main complication is calculating $\boldsymbol{C}_{obs}$ and $\boldsymbol{C}_{free}$, that are needed to determine the region where the robot can move without colliding. There are two main approaches to solve the planning problem: sampling-based and combinatorial motion planning.

In this project, a controller developed at the Institute for Artificial Intelligence[2] (IAI), *Giskard* (section 2.5.3), will be used as robot motion controller.

[2] http://ai.uni-bremen.de/

Motion planners have several ways of approaching the problem and finding a suitable trajectory for the robot to execute. Some algorithms are: sampling-based motion planning and optimization based motion planning.

2.4.1 Sampling-based motion planning

The planner samples different configurations in the C-space to construct collision-free paths. These paths are stored as 1D C-space curves. The main idea is to avoid the direct construction of the obstacle region C_{obs}. This means that instead of considering the obstacles directly, the planner uses a collision detector for each pose in the trajectory. This approach allows using the planner for a wide range of applications, one must only adapt the collision detector to the geometry of a specific robot.

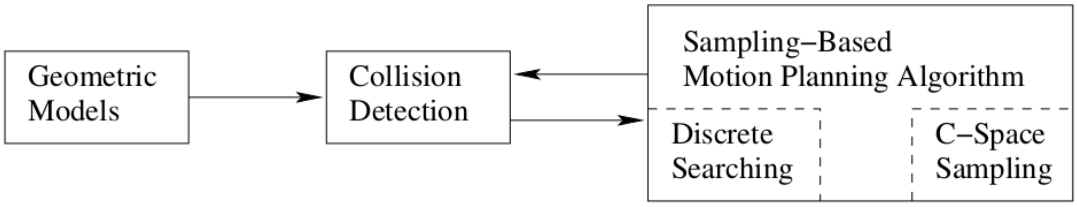


Figure 2.2: Sampling-based planning philosophy (LaValle, 2006, chap. 5, page 185).

The sampling-based planning, shown in figure 2.2, uses the collision detection as a "black box" between the motion planning and the geometric model of the robot. This generates an algorithm that is independent of the robot's geometry (LaValle, 2006) .

This "black box" approach can solve problems that involve thousands of geometric primitives representing the robot. It is practically impossible, according to Siliciano, Khatib (2007), to solve such problems with algorithms that uses the C_{obs} directly.

The disadvantage of these algorithms is that they don't assure to find a solution in a finite

amount of time. Combinatorial motion planning algorithms are able to return a solution, if it exists, in a finite amount of time.

A **rapidly-exploring random tree (RRT)** is a sampling-based planning algorithm that searches a collision-free path by randomly building a space-filling tree. It has a good performance and does not require any parameter tuning. As shown in figure 2.3, RRT reaches unexplored regions after just a few iterations.

Figure 2.3: Rapidly exploring random tree (LaValle, 2006, chap. 5, page 230).

The basic idea of RRT is to incrementally build a search tree in the C-space that tries to connect $\boldsymbol{q}_1$ and $\boldsymbol{q}_G$, avoiding the obstacles in the way, where $\boldsymbol{G}$ is the search graph. All the vertices of $\boldsymbol{G}$ are collision-free configurations.

The steps of this RRT algorithm for a C-space with no obstacles are shown in Algorithm 1 (LaValle, 2006). Let $\boldsymbol{S} \subset \boldsymbol{C}_{free}$ be the set of all points reached by $\boldsymbol{G}$. Each iteration, a new vertex is created and connected to the closest point in S along the shortest possible path.

If there are obstacles, the tree will go up to the obstacle's boundary, approaching as close as the collision detection algorithm allows.

The planner uses the obtained trees to find a path in $\boldsymbol{C}_{free}$ between the initial and the desired

Algorithm 1 Basic RRT

```
1: G.init(q_1);
2: for i = 1 to K do
3:     G.add_vertex(α(i));
4:     q_n ⟵ NEAREST(S(G, α(i)));
5:     G.add_edge(q_n, α(i));
6: end for
```

configuration. There are two different approaches for finding this path (figure 2.4):

- **Single-tree search:** The planner grows a tree from $\boldsymbol{q_1}$ as shown in figure 2.3 and checks in every step if it is possible to reach $\boldsymbol{q_G}$ with the $\boldsymbol{S}$ created by the tree.
- **Balanced, bidirectional search:** Instead of creating only one RRT, it creates an aditional tree that starts from $\boldsymbol{q_G}$, this is quite effective when obstacles are partially surrounding $\boldsymbol{q_G}$ or $\boldsymbol{q_1}$. The graph $\boldsymbol{G}$ is formed by two trees, T_a and T_b, growing from $\boldsymbol{q_1}$ and $\boldsymbol{q_G}$ respectively. Here, $\boldsymbol{S}$ is formed by T_a and T_b. After some iterations, the trees are swapped, so T_b is now growing around $\boldsymbol{q_1}$. Then, T_b connects to a new vertex created for T_a, this causes that T_b tries to grow towards T_a and vice-versa. The solution is found when both trees connect.

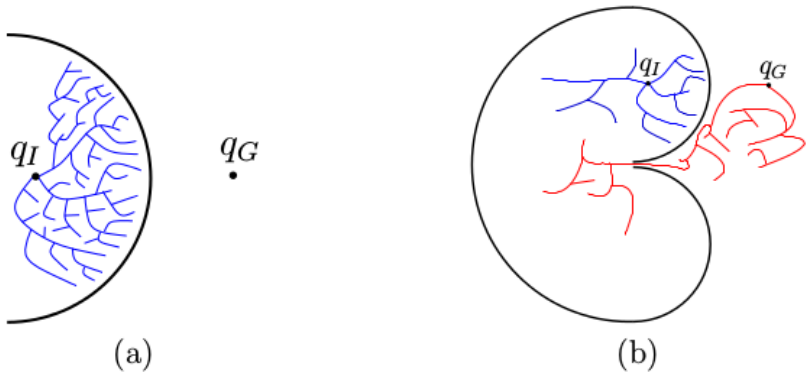


Figure 2.4: a) Single tree RRT, b) Bidirectional RRT (LaValle, 2006, chap.5, page 219).

The bidirectional search gives much better performance compared to the single-tree search.

2.4.2 Optimization-based motion planning

A different approach to solve the trajectory generation problem, is using an optimization algorithm that tries to minimize an error, in this case, the distance between the robot's end effector (EEF) an the goal.

The robot motion controller developed in this work uses a model predictive control (MPC) algorithm. As explained by Nikolaou (2001) *"all MPC systems rely on the idea of generating values for process inputs as solutions of an on-line (real-time) optimization problem"*.

MCP, also known as receding horizon control, is based on iterative optimization of a plant model. The approach used in this work, is solving the MPC with a sequence of convex quadratic programs (QP), where all motion constraints, goal and robot position are given as input to the optimization problem and joint velocities are obtained as result (section 3.3).

Quadratic programming is the process of solving a linearly constrained quadratic optimization problem of the form:

$$f(x) := \frac{1}{2}\vec{x}^T \mathbf{P}\vec{x} - \vec{x}^T b \tag{2.1}$$

The QP minimizes equation 2.1 over $\vec{x} \in \Re^n$ subject to the constraints:

$$\mathbf{C}\vec{x} = \vec{c}$$

$$\mathbf{D}\vec{x} \leq \vec{d}$$

Where the matrix $\mathbf{P} \in \Re^{n \times n}$ is symmetric positive definite. Matrices $\mathbf{C}$ and $\mathbf{D}$ are of size $\mathbf{C} \in \Re^{m \times n}$ and $\mathbf{D} \in \Re^{m \times n}$ and vector $\vec{b} \in \Re^n$. Vector $\vec{c} \in \Re^m$ sets the equality constraints and $d \in \Re^p$ the inequality constraints of the system (Hoppe, 2006). There are two approaches for solving this optimization problem, by active set strategies or by interior point methods.

Lets consider a discrete-time, time-invariant, linear system of the form:

$$\vec{x}_{k+1} = \mathbf{A}\vec{x}_k + \mathbf{B}\vec{u}_k \tag{2.2}$$

where $\vec{x}_k \in \Re^{n_x}$ is the system state, $\vec{u}_k \in \Re^{n_u}$, the inputs and $\mathbf{A} \in \Re^{n_x \times n_x}$ and $\mathbf{B} \in \Re^{n_x \times n_u}$ are fixed matrices. The constraints of input and state have to be guaranteed:

$$\underline{\vec{c}} \leq \mathbf{C}\vec{x}_k \leq \overline{\vec{c}} \tag{2.3}$$

$$\underline{\vec{d}} \leq \mathbf{D}\vec{u}_k \leq \overline{\vec{d}} \tag{2.4}$$

According to Ferreau et al. (2008), adapting equation 2.1 to this system, the optimization problem the MPC solves is:

$$min \;\; \vec{x}_N^T \, \mathbf{P}\vec{x}_N + \sum_{i=0}^{N-1} (x_i^T \; \mathbf{Q} \; x_i + u_i^T \; \mathbf{R} \; u_i) \tag{2.5}$$

subject to the constrains given by equations 2.3 and 2.4. The initial state x_0 is given as input to the first iteration of the QP. As result, the required input vector u_0 is obtained and applied to the system. The state achieved after applying u_0 is given as input to the next optimization problem. The MPC successively solves parametric QPs to obtain the required input sequence that lead to a desired system state.

Ferreau et al. (2008) developed a strategy for the fast solution of QPs called *online active set strategy*. They assume that the change in states between one QP and the next one is not big and therefore, the solution of the new QP is close to the solution of the old QP. This approach allows faster computations, that can be used in online applications of complex systems.

The software *qpOASES* implements this *online active set strategy* for use in MPC, therefore it was used in this work to calculate the joint velocities as explained in section 3.3.

One of the advantages of motion control based on QP *online active set strategy* is the computa-

tional speed. The trajectory can be generated fast enough for the system to recalculate it several times during the execution. Therefore, the system can react to unforeseen events, like a change in position of the goal. This is called *On-line Trajectory Generation*.

Using QP for robot motion control has already been done. Zhang, Zhang (2012) uses QP in a cyclic-motion generation scheme to prevent the angle drift of redundant robot manipulators. They apply the cyclic-motion criterion together with joint angle, velocity and acceleration limits to generate the QP, which is then solved using a neural network.

2.4.3 Other On-line Trajectory Generation approaches

The *Reflexxes Motion Libraries* (Kröger, 2011) are robot motion control libraries that provide tools for trajectory generation, the key capabilities of these libraries are the ability of calculating motions from an arbitrary initial state and the calculation of new motion on every control cycle.

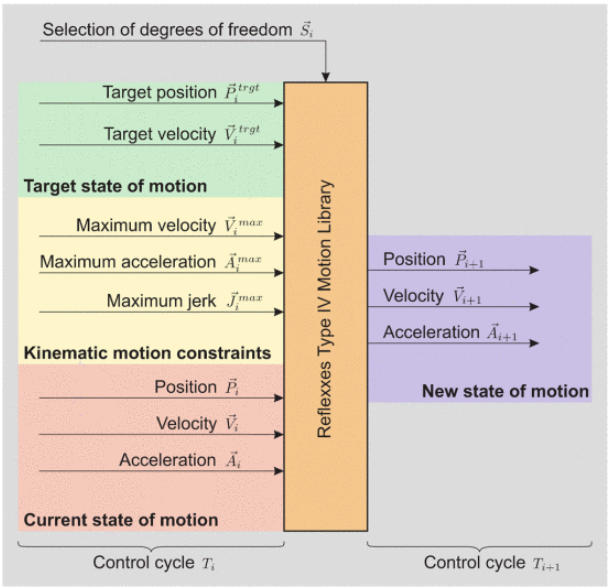


Figure 2.5: Interface of the Reflexxes Motion Libraries (Kröger, 2011, page 1).

Figure 2.5 shows the interface of the motion library, it considers the current state of the robot and it's constrains to calculate the new state the system needs to achieve the target position and

velocity. Doing the calculations on every control cycle (approx 1 msec) allows the system to react to unforseen events, such as switching of coordinate frames, and to sensor signals.

Reflexxes is based on Kröger, Wahl (2010), they propose a motion control technique that allows instantaneous switching between *trajectory-following control* and *sensor-guided control*.

2.5 Software

In the last years, the number of applications where humans and robots work in proximity to each other has increased. Robot control has become a wide-spread research area. This has lead to the development of many software programs used for robot manipulation.

The Institute for Artificial Intelligence (IAI) uses ROS to communicate with the robots. Since the system developed during this thesis is designed for the Boxy robot of the IAI, it was done using ROS as base. For trajectory planning, I developed a controller based on Giskard (section 2.5.3), with RVIZ as a side tool for visualization. The collision detection of the system uses Moveit! (section 2.5.6). I also developed a small perception system for detecting the objects the robot will grasp. This was done using the Chilitags fiducial markers (section 2.5.4), markers are placed on each object to identify them and determine their position and orientation.

2.5.1 ROS

Robot Operating System (ROS) is an open source software that helps developers to create robot applications. According to the official documentation[3], ROS is a meta-operating system, a system that handles other operating systems, it handles hardware abstraction, low-level device

[3] `http://wiki.ros.org/ROS/Introduction`

control, implementation of commonly-used functionality, message-passing between processes, and package management. ROS also provides a wide range of libraries and tools for robots, such as drivers and visualizers. Some of these libraries are focused on mobility, manipulation and perception tasks.

One of the ROS main components is its communication infrastructure. This has a message passing interface that provides inter-process communication and is seen as a middleware[4].

The communication consists of a publish/subscribe message passing system, where one program publishes information under a certain topic. All other programs subscribed to this topic will receive the information. A program can be sending information through one or more topics and, at the same time, receiving information from other topics. ROS works as the middleware that manages and distributes these messages.

ROS uses *nodes* to do computations. A node is a process that communicates with other nodes using topics and the parameter server. The main reason behind nodes is the simplification of program codes. A robot is controlled using many nodes, each one in charge of a small task, such as localization or controlling a wheel; this breaks down the complexity of the system into many smaller subsystems (modules) that work independent of each other, communicating through topics and parameters.

Since most applications require several nodes to control all the subsystems, it is useful to have a code that can launch multiple nodes locally and remotely. A *launch file* is a XML configuration file with a *.launch extension, it can specify a set of parameters and nodes to launch. These files can also include other launch files.

[4] `http://www.ros.org/core-components/`

2.5.2 RViz

RViz is a 3D visualization tool for ROS. RViz displays the sensor data and the robot's state information taken from ROS. Using RViz, we can visualize the current state of the robot (figure 2.6), visualize simulated trajectories, and display information from the sensors, such as 3D point-clouds from the Kinect or the image obtained by a camera.

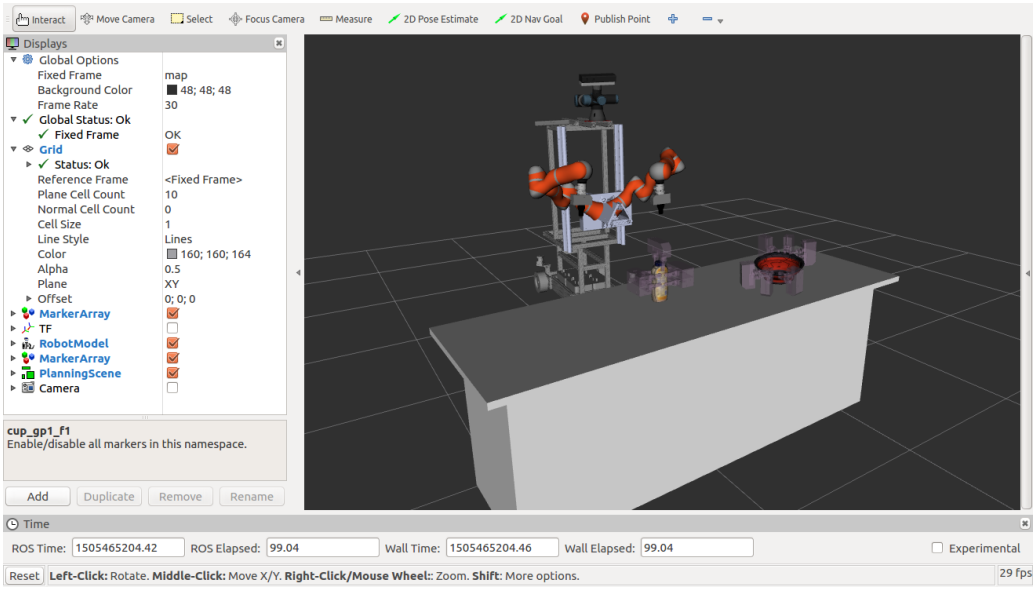


Figure 2.6: RViz visualization of Boxy.

RViz uses a *fixed frame* as a static reference for the visualization, the default frame is `\world`. All the information that RViz receives is displayed with respect to this frame, including: the robot's current state, location of objects in the environment, and data from sensors.

Figure 2.6 shows a simulation of the Boxy robot detecting several objects located on a table in front of it, the transformation frames (TF) of both end effectors (EEF) can be seen.

2.5.3 Giskard

Giskard[5] is a constraint- and optimization-based framework (section 2.4.2) for robot motion control developed by the Institute for Artificial Intelligence[6]. It uses Quadratic Programming (QP) for solving the motion control problem (section 2.4.2). This controller can react to unexpected events, since it calculates the next trajectory step on every control cycle (Fang et al., 2016). Giskard was used as base to develop the motion controller used in this work (section 3.3), which also uses QP to generate a trajectory.

This controller allows using action models that inform the robot how to execute a specific motion, such as pouring or grasping. The *giscard_boxy*[7] package contains files with specific configuration for using the *giskard* motion controllers of the Boxy robot, which is the robot used in this thesis.

2.5.4 Chilitags

Chilitags[8] is a software library for the detection and identification of 2D fiducial markers for robotics and augmented reality. Fiducial markers are objects placed in the view field of a perception system to be used as a reference. In other words, *Chilitags* are markers added to objects used as reference points in a perception system. In this case, the markers help not only to find the relative position and orientation of the objects with respect to the camera, but also to identify them.

[5] https://github.com/SemRoCo/giskard_core
[6] http://ai.uni-bremen.de/
[7] https://github.com/SemRoCo/giskard_boxy
[8] http://chili.epfl.ch/software

(a) Tag 0 (b) Tag 1 (c) Tag 2

Figure 2.7: Example of Chilitags fiducial markers, each tag has a different ID number.

Figure 2.7 shows three fiducial markers that were placed in different objects. The precise size of the markers is known beforehand, so when placed in front of a camera, the perception system calculates the exact distance to the marker and identifies it using the pattern.

In this project, perception was done by attaching markers to several kitchen objects the robot had to grasp and detecting the using a Microsoft Kinect 2 placed on top of the robot (see figure 1.2).

2.5.5 Naive Kinematic Simulator

The *iai_naive_kinematic_simulator*[9] is a kinematic simulator for robots based on ROS. It was developed by the Institute for Artificial intelligence.

It is a light-weight that does not consider inertia or friction, therefore kinematics algorithms can use it for fast calculations. In this thesis, it is used to simulate the commands generated by the motion controller and generate a trajectory (section 3.3).

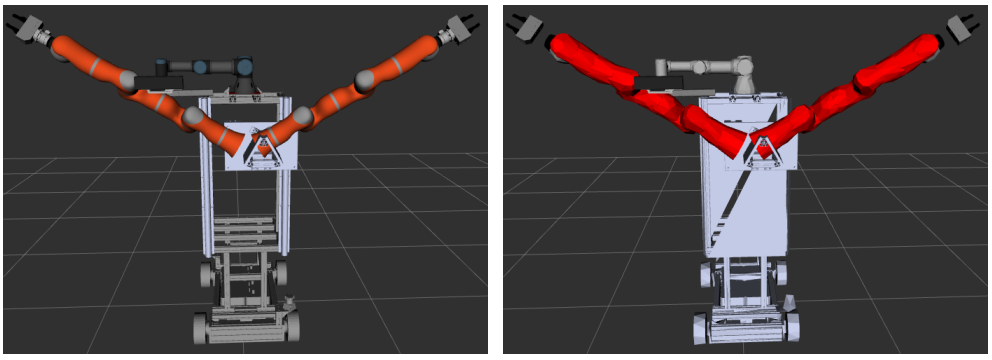
[9] https://github.com/code-iai/iai_naive_kinematics_sim

2.5.6 MoveIt!

In Chitta et al. (2012), MoveIt! is defined as "*a set of software packages integrated with the Robot Operating System (ROS) and designed specifically to provide such capabilities (avoid collisions with humans and other obstacles), especially for mobile manipulation.*".

MoveIt! Planning Scene[10] is able to create an environment where the robot and external objects can be loaded. These objects can be seen as obstacles or as objects the robot will interact with. Moveit! represents the robot using a kinematic description of it, an URDF (section 2.6). The environment is described using an Octomap; an Octomap is a probabilistic 3D mapping framework based on octrees, it provides a 3D occupancy grid that defines which parts of the environment are free.

It is important to mention that MoveIt! uses 2 models for each object, as shown in figure 2.8, one for visualization and one for collision.



(a) Visualization Model (b) Collision Model

Figure 2.8: Models of Boxy used by MoveIt!

The *visualization model* is used by RViz (section 2.5.2) to visualize the robot in the computer. This model is normally as close as possible to the real robot, that means the geometry tends to

[10] http://docs.ros.org/indigo/api/moveit_core/html/classplanning__scene_1_1PlanningScene.html

be complex and the files containing it are heavy. We use this model for visualizing the planned trajectory before executing it, or for visualizing the current state of the robot.

The *collision model* of Boxy contains simplified geometry of the robot, it is usually slightly bigger than the real robot for security reasons. The collision model is the one MoveIt! uses to check for collisions. Since the collision check is done for every pose in the trajectory and for every part in the model, it is convenient to have simplified geometry to speed up the calculations.

2.6 Robot Description

In oder to calculate the position and movements of a robot, all it's elements must be represented in a way that the robot controller and the trajectory planner understands them. This implies models or descriptions where the geometry, movement ranges and kinematics are described. Such description is made with a Unified Robot Description Format (URDF).

As the name implies, a URDF is a file used to describe a robot, it contains XML specifications of the robot's geometry, kinematics, inertial, and sensing properties. It is the native robot description format in ROS. The URDF loads meshes that contains the geometry of one or several robot links and describes it's relative position with respect to the previous link, it also specifies the type and range of movement it has.

Instead of writing a URDF file, programmers normally write a XACRO of the file, a XACRO is a XML macro, which is easier to read and allows code reuse.

CHAPTER 3

Methodology

This chapter describes the general structure of the system developed in this work and all the elements that conform it. At the beginning, the perception system as well as the object's models are explained. A detailed description of the optimization-based motion controller can be found in section 3.3, where several examples of basic controllers are presented. Afterwards, the controller developed for the robot is explained. In the end, the collision detection system is described.

3.1 System description

The system developed in this work, which has the objective of allowing the robot to identify and grab a specified object, comprises several parts:

- **Object Database:** A database with a 3D model of the objects the robot will be able to grasp, as well as predefined grasping poses for each object.
- **Perception system:** In order to simplify the perception task, all objects are identified using markers. The position and orientation of the markers with respect to the reference frame of the objects is known.

- **Motion controller:** A constraint-based, optimization-based controller was developed. This controller allows the parametrization of the trajectory and automatically selects the best grasping pose according to the initial conditions.

- **Collision detection:** Since the motion controller is not able to detect collision objects in the environment, it is important to evaluate if the obtained trajectory would generate a collision

A general description of the system's architecture is shown in figure 3.1. The system uses ROS (section 2.5.1) to send and receive information from the robot and the user, each rectangle depicts a ROS node. All nodes communicate using ROS services, actions and messages.

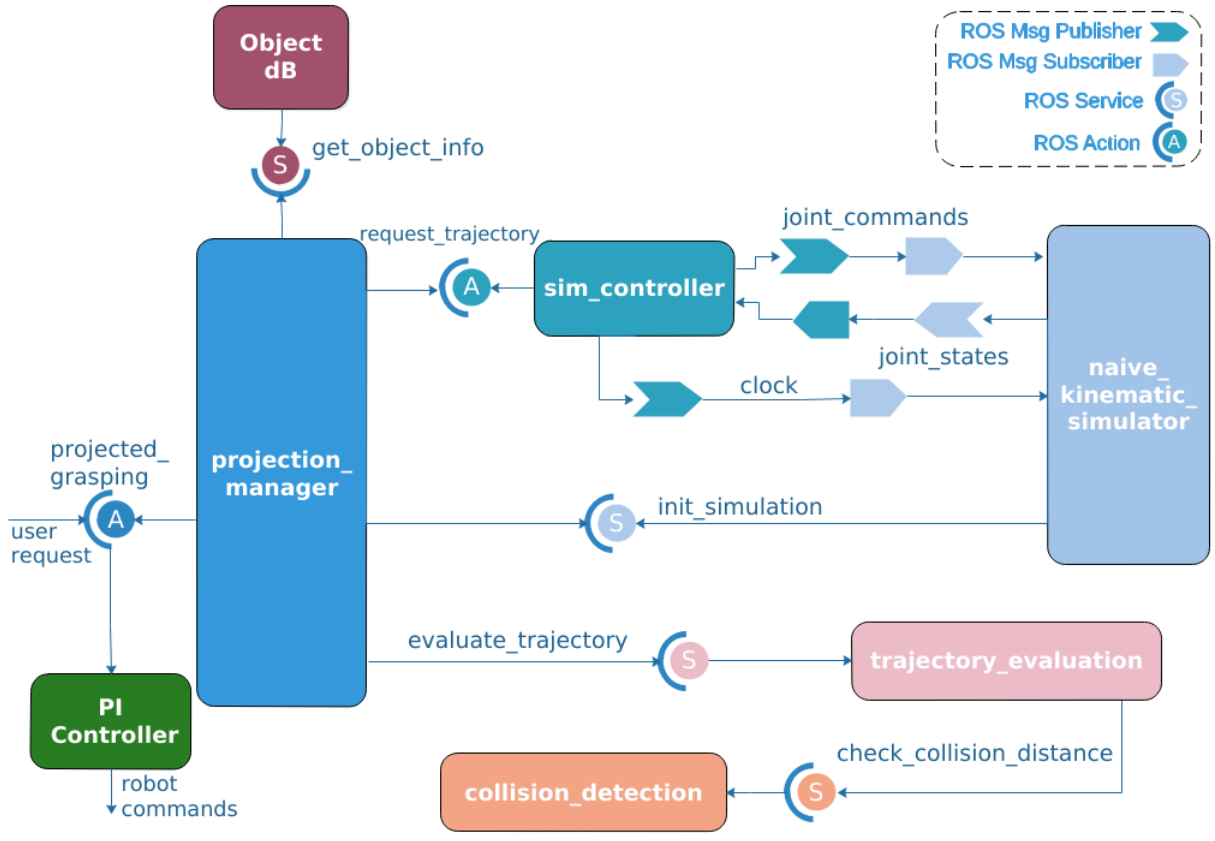


Figure 3.1: System architecture: The perception system (*Object_dB*) sends location of found objects to the system manager (*projection_manager*), which requests a trajectory to the controller (*sim_controller*).

The *Object dB* node is in charge of detecting objects and publishing its current location, as well as the defined grasping poses for each object. The object's information is read from a YAML file (section 3.2.1). This node provides a service that publishes a list of found objects.

The component in charge of generating the trajectory is *sim_controller*, it receives the goal pose, calculates the required velocity for each joint, and sends it to the *naive_kinematic_simulator*[1], which simulates the robot's movements. It uses the controller described in section 3.3.

The *projection_manager* node coordinates the whole system. It receives the request to grasp an object, asks the *Object dB* if the object was detected and requests the generation of a trajectory to the motion controller.

The *projection_manager* requests several trajectories to the *sim_controller* to have several options to choose from. It calculates the distance from both EEF to all the object's grasping poses. Based on the distance and the initial manipulability of each arm, selects the most suitable grasping pose (GP) and requests a trajectory to it, then discards this GP and selects the second best one and gets another trajectory. This process is repeated a couple of times to ensure that trajectories reaching different goals with both arms are obtained.

Then, *projection_manager* sends the trajectories to the *trajectory_evaluation* service (section 3.4), which evaluates the trajectories, selects one and sends it back as result.

The *trajectory_evaluation* service starts by sending each trajectory to the *collision_detection* service. If a collision is found, the trajectory is discarded. The remaining trajectories are scored based on length, smoothness, minimum distance to a collision object and the manipulability of the arm after reaching the goal.

[1] https://github.com/code-iai/iai_naive_kinematics_sim

Once a trajectory is selected, the *PI Controller* sends it to the robot and ensures that it follows the trajectory.

3.2 Perception System

The system requires previous knowledge about the objects the robot will interact with, it must be able to recognize the object and identify possible ways to grasp it. A 3D model of all objects was created, figure 3.2 shows an example of the models. These models are used for visualization in RVIZ.

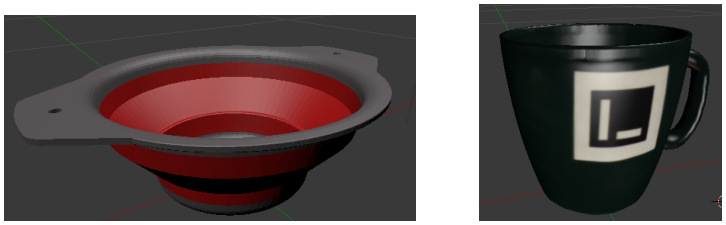


Figure 3.2: DAE models of a bowl and a cup. On the cup, one of the markers used for perception can be seen.

In order to simplify the perception task, markers were added to all objects. The markers used are *Chilitags*[2], which can be integrated with ROS[3]. Once the tags or markers are detected by the camera, its position is published to a ROS topic. The *Object dB* node matches the corresponding objects to the found markers and displays the models in RVIZ.

3.2.1 Object Database

All the information about the objects is stored in a YAML file that works as database. The information is stored using the following structure:

[2]http://chili.epfl.ch/software
[3]https://github.com/chili-epfl/ros_markers

```
object_name:
  id: 01                    # object ID number
  marker: [tag_1, tag_2]    # markers placed in the object
  tag_1:                    # Position and orientation of the markers with
                            # respect to the object frame of reference
    position: [0.003, 0.069, 0.03]
    orientation: [ 0.670,  0.016, -0.015,  0.741]
  tag_2:
    position: [0.005, 0.0678, 0.0345]
    orientation: [ 0.061,  0.740, -0.665,  0.068]
  mesh: package://iai_markers_tracking/meshes/object.dae  # 3D model
  mesh_collision: package://iai_markers_tracking/meshes/collision.stl
  # Use collision model always in meters
  scale: 1.0                # if the model is in cm = 1, m = 0.01
  gripper_opening: 0.0055   # how much the gipper should open to
                            # grasp the object
  grasping_poses:           # predifined grasping poses
    - p_id: gp1
      position: [0.058, 0.0, 0.048]
      orientation: [-0.70710678, 0.0, 0.0, 0.70710678]
    - p_id: gp2
      position: [0.058, 0.0, 0.048]
      orientation: [0.70710678, 0.0, 0.0, 0.70710678]
    - p_id: gp3
      position: [-0.036, -0.0, 0.06]
      orientation: [0, 1.0, 0.0, 0.0]
```

This structure is filled for every object the robot will grasp and added at the end of the YAML file. The `mesh` field contains the visualization model of the object, it can contain texture and colors. The `mesh_collision` field contains a simplified model of the object (section 2.5.6), a simpler mesh that is used during collision detection.

3.2.2 Grasping Poses

For humans, it is intuitive how an object can be grasped, but for a robot it is not a simple task. In this project, grasping poses (GP) are already defined for each object, so that the robot only has to choose which pose to use in every situation.

Figure 3.3: 3D model of a tomato sauce package with two grasping poses. The object and GP have reference frames.

Figure 3.3 shows an object with two predefined GPs. The perception system detects the position of the *Chilitags* markers, searches the database for the corresponding object and GP and displays them in RVIZ, it also publishes a TF for both.

3.3 Motion Controller

The motion control system used to generate and control the movement of the robot is based on Giskard (section 2.5.3), an optimization-based controller. The controller developed in this work, as well as Giskard, uses *qpOASES*[4] to calculate a trajectory for each joint. A brief explanation of optimization-based controllers can be found in section 2.4.2.

As stated by Ferreau et al. (2014), qpOASES is an open-source optimization software that can solve quadratic programming (QP) problems of the form:

$$\min_{\vec{x}} c(\vec{x}) \;=\; \min_{\vec{x}} \left(\frac{1}{2}\, \vec{x}\,' \,\mathbf{H}\, \vec{x} + \vec{x}\,' \,\vec{g}(\omega_0) \right) \tag{3.1}$$

Where $H \in \Re^{nV \times nV}$ is a symmetric and positive (semi-) definite Hessian matrix and $\vec{g} \in \Re^{nV}$ is a gradient vector that depends on the parameter ω_0, with nV being the length of the vector $\vec{x}$. The software tries to minimize the cost function $c(x)$. This minimization is subject to the

[4] `https://projects.coin-or.org/qpOASES`

following constraints:

$$\vec{lb}(\omega_0) \leq \vec{x} \leq \vec{ub}(\omega_0)$$

$$\vec{lbA}(\omega_0) \leq \mathbf{A}\vec{x} \leq \vec{ubA}(\omega_0)$$

where $\vec{lb}$, $\vec{ub} \in \Re^{nV}$ are the lower and upper constraint bound vectors of $\vec{x}$, the matrix $\mathbf{A} \in \Re^{nC \times nV}$ is the constrain matrix with the lower and upper constraint boundary vectors $\vec{lbA}$ and $\vec{ubA} \in \Re^{nV}$. Equality constraints can be achieved by setting equal upper and lower boundaries.

Several simple controllers were built and tested in order to understand the characteristics of the trajectories obtained using qpOASES. These controllers describe a two degree of freedom (DOF) linear manipulator. A state vector, $\vec{s} \in \Re^{nV}$, that describes the system is used as the variable the cost function minimizes:

$$\vec{s} = [\, \dot{q}_0 \;\; \dot{q}_1 \;\; \varepsilon]^T \tag{3.2}$$

where $\dot{q}_0$ is the velocity of the first joint, $\dot{q}_1$ the velocity of the second joint, and ε is a slack variable. Introducing this vector in the cost function (eq. 3.1) and setting the gradient $\vec{g}$ to zero:

$$\min_{\vec{s}} c(\vec{s}) = \min_{\vec{s}} \left(\frac{1}{2}\, \vec{s}' \, \mathbf{H} \, \vec{s} \right) \tag{3.3}$$

with the minimization constraints given by:

$$\vec{lb}(t) \leq \vec{s} \leq \vec{ub}(t) \tag{3.4}$$

$$\vec{lbA}(t) \leq \mathbf{A}\vec{s} \leq \vec{ubA}(t) \tag{3.5}$$

We define different weights for each parameter of the state vector $\vec{s}$ by setting the $\mathbf{H}$ matrix to:

$$\mathbf{H} = diag(\vec{\omega})$$

with the weight vector $\vec{\omega} \in \Re^{nV}$. These weights allow us to prioritize the movement of one specific joint over the others, considering that the cost function for the 2 DOF controller from

eq 3.3 is given by:

$$c(\vec{s}) \;=\; \frac{1}{2}\,\vec{s}'\,\mathbf{H}\,\vec{s} = \;\frac{1}{2}\,(\,\dot{q_0}^2\omega_1 + \dot{q_1}^2\omega_2 + \varepsilon^2\omega_3) \tag{3.6}$$

The software will try to minimize the cost of solving the problem by setting a higher value for the variables with a lower weight.

The result of this minimization problem is the velocity required by each joint to reach a goal position in a certain time. This can be set as an equality constraint:

$$error \;=\; q_{eef,\,des} - q_{eef} \tag{3.7}$$

$$error \;\leq\; \dot{q}_0 + \dot{q}_1 + \varepsilon \;\leq\; error \tag{3.8}$$

where $q_{eef,\,des}$ is the desired position of the end effector (EEF) and q_{eef} the current position of the EEF. The position error can be used as a velocity goal if we suppose that we want to cover the distance error in one time unit. Velocity constraints are also established for each joint:

$$-\dot{q}_{0,max} \;\leq\; \dot{q}_0 \;\leq\; \dot{q}_{0,max} \tag{3.9}$$

$$-\dot{q}_{1,max} \;\leq\; \dot{q}_1 \;\leq\; \dot{q}_{1,max} \tag{3.10}$$

The software will try to find the velocity required by each joint to correct the position error in one time step. Since the joints have velocity limits, it might not be possible to set velocities that satisfy the constraint set by eq 3.8 only using the joint velocities, that is why the slack variable ε was introduced, with the constraints:

$$-\varepsilon_{max} \;\leq\; \varepsilon \;\leq\; \varepsilon_{max} \tag{3.11}$$

where $\varepsilon_{max} \gg q_{0,max},\, q_{1,max}$. The weight of the slack value is higher than the ones of the joint velocities, so the software will try to give a higher value to the velocities than to the slack variable.

The joint limits also have to be taken into consideration:

$$(-q_{0,max} - q_0) \leq \dot{q}_0 \leq (q_{0,max} - q_0) \tag{3.12}$$

$$(-q_{1,max} - q_1) \leq \dot{q}_1 \leq (q_{1,max} - q_1) \tag{3.13}$$

These constraints will reduce the joint velocities when they are approaching the limits until they reach zero at joint limit. The obtained velocity is the velocity the joints would require to reach the goal position in one time instant. However, due to physical limitations, joints can not immediately reach maximum velocity in one time step, so acceleration limits have to be established:

$$-a_{0,max} \leq \dot{q}_0 \leq a_{0,max} \tag{3.14}$$

$$-a_{1,max} \leq \dot{q}_1 \leq a_{1,max} \tag{3.15}$$

The maximum acceleration is calculated on every iteration using

$$a_{n,max\_lim} = \frac{\dot{q}_{n,max} - a_{n,max}}{\dot{q}_{n,max}} + a_{n,max} \tag{3.16a}$$

$$a_{n,min\_lim} = \frac{\dot{q}_{n,max} - a_{n,max}}{\dot{q}_{n,max}} - a_{n,max} \tag{3.16b}$$

where n is the joint number and $a_{n,max}$ a constant acceleration limit. The acceleration constraint is calculated based on the current joint speed.

Combining equations 3.8 to 3.15 to create the constraint bound vectors from equations 3.4 and 3.5, we obtain

$$\begin{pmatrix} -\dot{q}_{0,max} \\ -\dot{q}_{1,max} \\ -\varepsilon_{max} \end{pmatrix} \leq \begin{bmatrix} \omega_1 & 0 & 0 \\ 0 & \omega_2 & 0 \\ 0 & 0 & \omega_3 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon \end{pmatrix} \leq \begin{pmatrix} \dot{q}_{0,max} \\ \dot{q}_{1,max} \\ \varepsilon_{max} \end{pmatrix} \tag{3.17}$$

and

$$\begin{pmatrix} error \\ -q_{0,max} - q_0 \\ -q_{1,max} - q_1 \\ a_{0,min\_lim} \\ a_{1,min\_lim} \end{pmatrix} \leq \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon \end{pmatrix} \leq \begin{pmatrix} error \\ q_{0,max} - q_0 \\ q_{1,max} - q_1 \\ a_{0,max\_lim} \\ a_{1,max\_lim} \end{pmatrix} \tag{3.18}$$

The constrains set by these equations are the groundwork for the basic controllers explained in the following subsections. All these cases simulate a translational 2 DOF system, the goal is to place the EEF at a given position (goal position). The initial position of each joint is given, as well as a goal for the EEF. In these examples, the initial velocity and acceleration of the joints is assumed to be zero, but other initial conditions can be set.

3.3.1 Simple 2 DOF controller

A proportional gain, p, for the joints' goal is introduced in order to reduce the number of iterations the program requires to solve the problems. In other words, the proportional gain can be used to increase the joint velocities. This gain is added to equation 3.7:

$$error = p * \left(q_{eef,des} - q_{eef}\right) \tag{3.19}$$

Figures 3.4 and 3.5 show the position and velocity of a 2 DOF manipulator, whose trajectory is calculated using equations 3.17 and 3.18 to obtain the joint velocity in each time step. Figure 3.4 shows the EEF position (dashed), velocity and the goal position (dotted), while figure 3.5 shows the position and velocity of the joints q_0 and q_1.

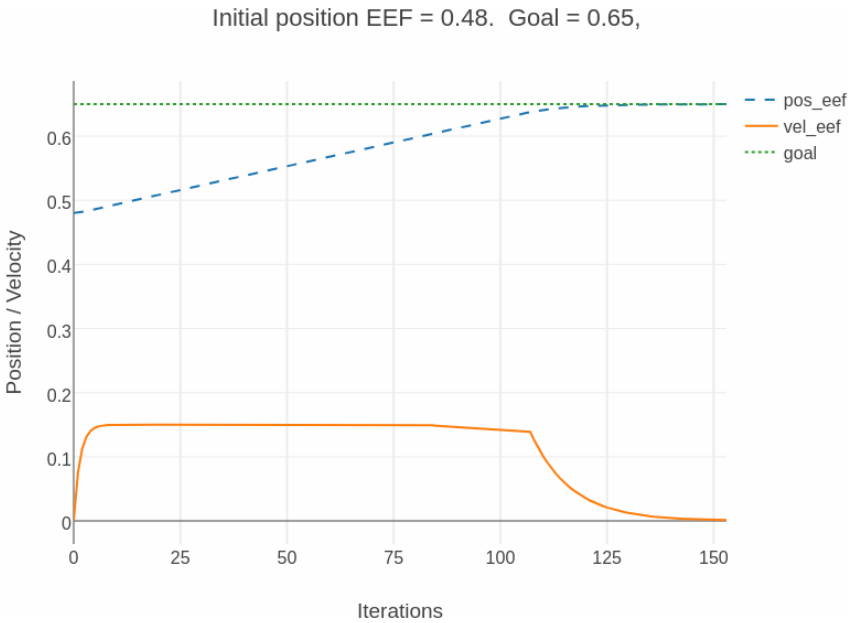
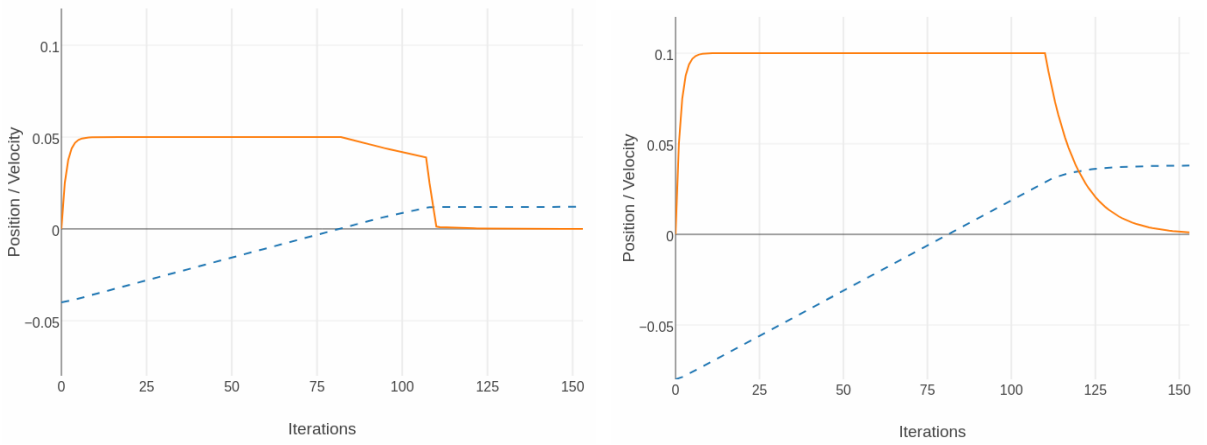


Figure 3.4: EEF trajectory. Initial acceleration is limited by constraints. Velocity decreases as EEF approaches the goal position.



(a) Joint q_0 (b) Joint q_1

Figure 3.5: Joints trajectories (dashed). Each joint has different velocity (continuous line) and acceleration constraints.

It can be seen that the velocity decreases once the EEF is approaching the desired position, also the joints do not reach the maximum velocity in the first time step. This is due to the acceleration constraint given by equation 3.16.

3.3.2 Joints goal and EEF goal

This controller allows to specify a goal position for one of the joints as well as for the EEF. Figures 3.6 and 3.7 show the behavior of the EEF and both joints while reaching the specified goals. Each joint has different velocity and acceleration constraints.

In this controller, the weight vector is defined as $\vec{\omega} = [\omega_1,\ \omega_2,\ \omega_3,\ \omega_4]$. Where ω_1 and ω_2 are the joint weights, ω_3 is the weight of the slack factor and ω_4 is the weight of the joint goal. Equation 3.18 is rewritten as

$$\begin{pmatrix} error \\ -q_{0,max} - q_0 \\ -q_{1,max} - q_1 \\ a_{0,min\_lim} \\ a_{1,min\_lim} \\ joint\_error \end{pmatrix} \leq \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon \\ \varepsilon_j \end{pmatrix} \leq \begin{pmatrix} error \\ q_{0,max} - q_0 \\ q_{1,max} - q_1 \\ a_{0,max\_lim} \\ a_{1,max\_lim} \\ joint\_error \end{pmatrix}$$

where ε_j is the slack factor for the joint goal and $joint\_error$ is the current error of the joint q_0 to the joint goal, given by $joint\_error = q_{0,des} - q_0$.

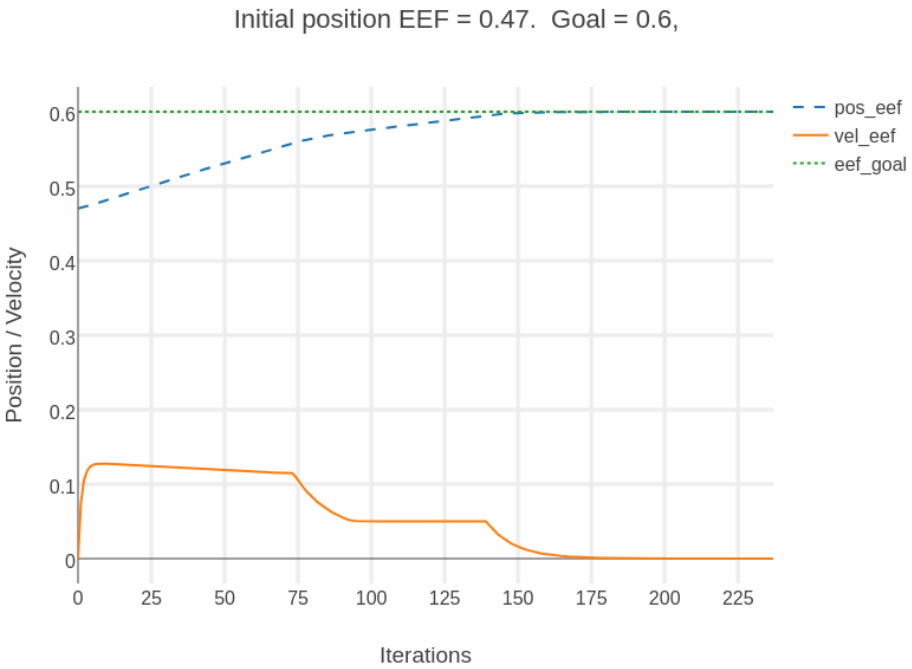
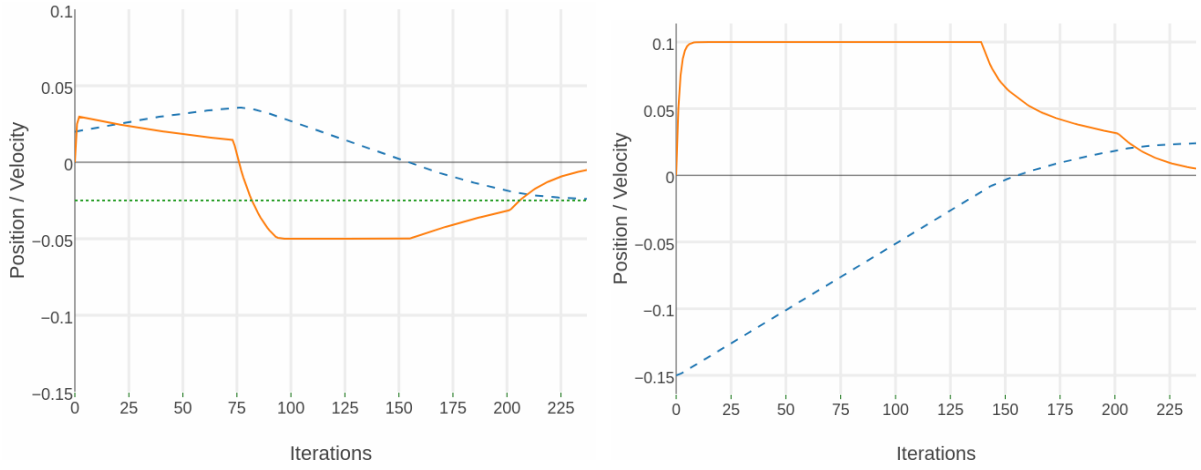


Figure 3.6: Trajectory (dashed), velocity (continuous) and goal (dotted) of the EEF.

The initial position of the EEF is 0.47, and of the joints $q_0 = 0.02$ and $q_0 = -0.15$, respectively. Only a goal for the EEF and for the first joint, $EEF_f = 0.6$ and $q_{0f} = -0.25$ are specified.

It can be seen in figure 3.7 that one of the joints changes direction after around 75 iterations. This happens because the goal of the EEF and of the joint are in different directions, the system tries to minimize first the EEF error and afterwards the one of the joint.



(a) Joint q_0 (b) Joint q_1

Figure 3.7: Trajectories (dashed) and velocities of both joints and goal (dotted) of q_0.

This behavior suggest that it might be better to split the problem when goals for the joints are specified, where in the first part the joint goals are satisfied and kept constant and in the second part the EEF is reached.

3.3.3 Position range as goal for the EEF

In this example, the goal specified for the EEF is not one specific position, but a range, where any point in this area is a valid goal for the EEF.

Figure 3.8 shows the trajectory followed by the EEF while reaching the goal, it stops shortly after entering the given goal range. If a specific point inside this range is preferred as goal, an *attractor* can be created,so that the EEF will try to reach the point defined by the attractor, if it is not possible, it will stay in the goal range.

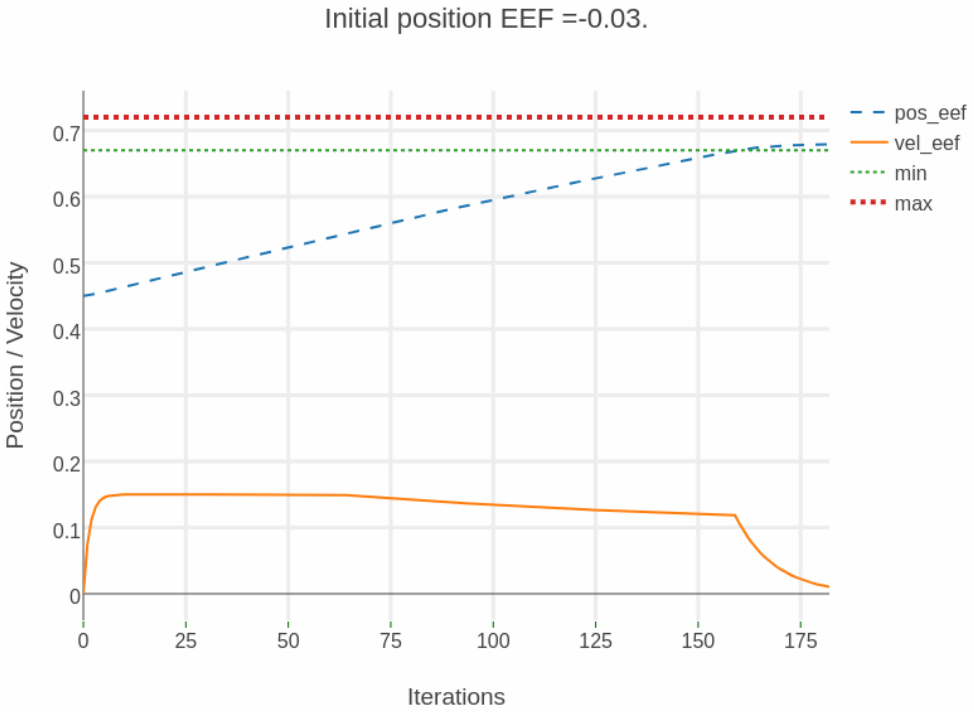


Figure 3.8: Trajectory (dashed) and velocity of the EEF, the goal range is between the dotted lines.

To achieve this behavior, matrix **A** and it's limits, $\vec{lb}A$ and $\vec{ub}A$, are modified, rewriting equation 3.18 as:

$$\begin{pmatrix} error \\ -q_{0,max} - q_0 \\ -q_{1,max} - q_1 \\ a_{0,min\_lim} \\ a_{1,min\_lim} \\ range_{min} - q_{eef} \end{pmatrix} \leq \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon \\ \varepsilon_r \end{pmatrix} \leq \begin{pmatrix} error \\ q_{0,max} - q_0 \\ q_{1,max} - q_1 \\ a_{0,max\_lim} \\ a_{1,max\_lim} \\ range_{max} - q_{eef} \end{pmatrix}$$

where ε_r is the slack factor for the goal range. The error is calculated using the attractor (goal) $error = attr - q_{eef}$. Here, the weight vector is defined as $\vec{\omega} = [\omega_1,\ \omega_2,\ \omega_3,\ \omega_4]$. The constraints of the state vector (eq. 3.17) is rewritten as

$$\begin{pmatrix} -\dot{q}_{0,max} \\ -\dot{q}_{1,max} \\ -\varepsilon_{max} \\ \varepsilon_{r,max} \end{pmatrix} \leq \begin{bmatrix} \omega_1 & 0 & 0 & 0 \\ 0 & \omega_2 & 0 & 0 \\ 0 & 0 & \omega_3 & 0 \\ 0 & 0 & 0 & \omega_4 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon \\ \varepsilon_r \end{pmatrix} \leq \begin{pmatrix} \dot{q}_{0,max} \\ \dot{q}_{1,max} \\ \varepsilon_{max} \\ \varepsilon_{r,max} \end{pmatrix}$$

where ω_1 and ω_2 are the joint's weights, ω_3 is the weight of the attractor, and ω_4 is the weight of the goal range.

Figures 3.9 to 3.10 show the behavior of the EEF and joints. In this example, the attractor was set at 0.7 with the range between 0.67 and 0.72. It can be seen how the EEF reaches the goal set by the attractor, instead of just staying at the range borders.

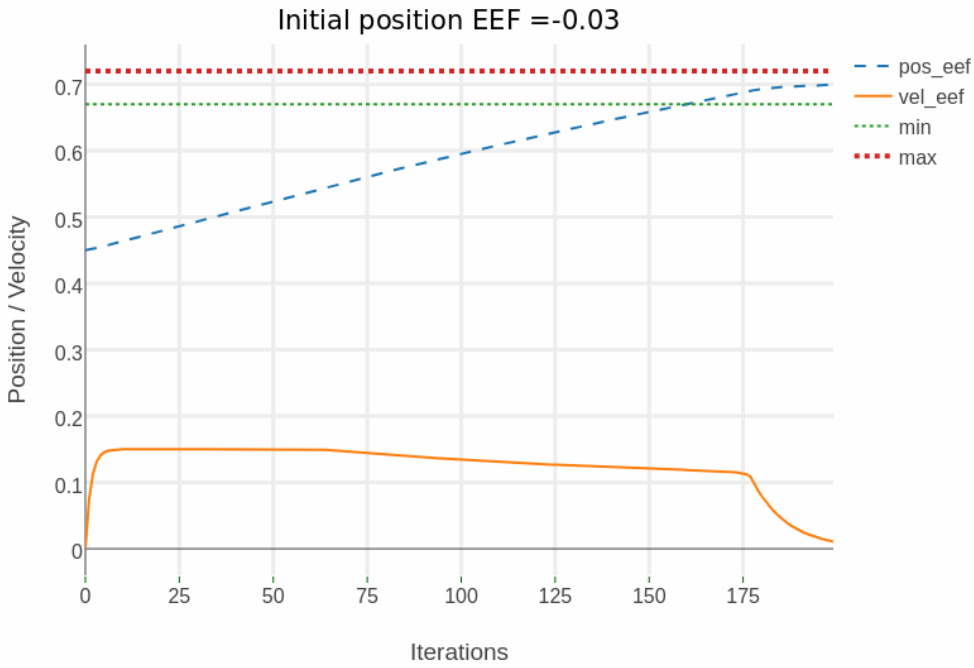
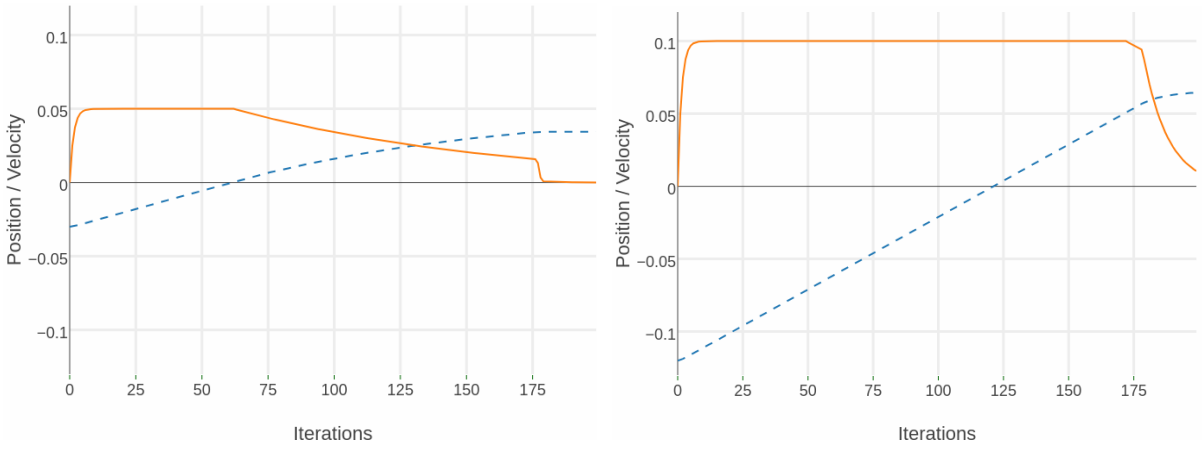


Figure 3.9: Trajectory (dashed) and velocity of the EEF, here an attractor is defined inside the range.



(a) Joint q_0 (b) Joint q_1

Figure 3.10: Trajectory (dashed) and velocity of both joints. Each joint has a different velocity and acceleration constraint, as well as different joint limits.

The weight of the attractor is set higher than the range's weight, this means that the cost of reaching the point defined by the attractor is higher than just going inside the range. By doing this, the optimization algorithm prioritizes moving the EEF to the given range. Modifying these weights determine how close will the EEF come to the attractor.

3.3.4 Multiple goals for the EEF

It is also possible to specify multiple goals for the EEF and select one of them using the weights assigned to each goal. This controller shows a two DOF linear manipulator where two goals are specified. In this case, the matrix **A** and it's limits (given by equation 3.18) are given by:

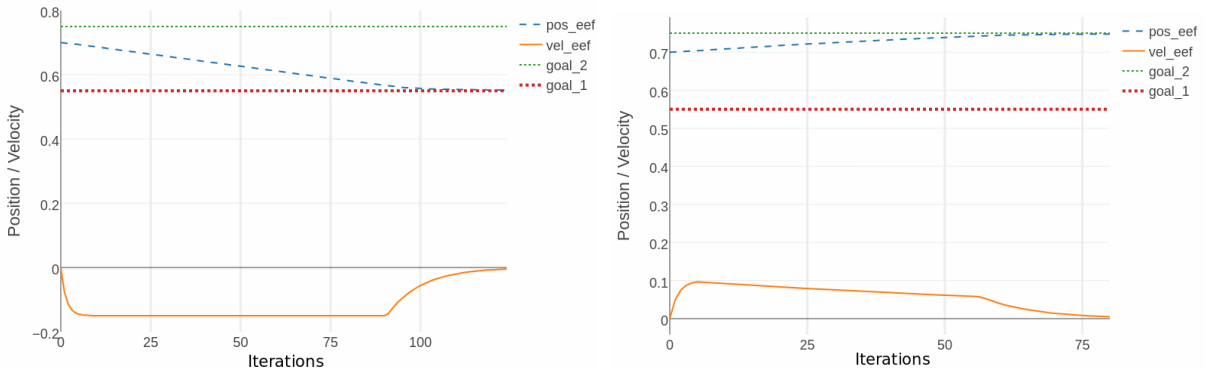
$$\begin{pmatrix} error_{goal1} \\ error_{goal2} \\ -q_{0,max} - q_0 \\ -q_{1,max} - q_1 \\ a_{0,min\_lim} \\ a_{1,min\_lim} \end{pmatrix} \leq \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} \leq \begin{pmatrix} error_{goal1} \\ error_{goal2} \\ q_{0,max} - q_0 \\ q_{1,max} - q_1 \\ a_{0,max\_lim} \\ a_{1,max\_lim} \end{pmatrix}$$

where ε_1 and ε_2 are the slack factors for the first and second goal, respectively. The weight vector is defined as $\vec{\omega} = [\omega_1,\ \omega_2,\ \omega_3,\ \omega_4]$. Where ω_1 and ω_2 are the joint's weights, ω_3 and ω_4 are the goals' weight. The constraints of the state vector (eq. 3.17) is rewritten as:

$$\begin{pmatrix} -\dot{q}_{0,max} \\ -\dot{q}_{1,max} \\ -\varepsilon_{1,max} \\ \varepsilon_{2,max} \end{pmatrix} \leq \begin{bmatrix} \omega_1 & 0 & 0 & 0 \\ 0 & \omega_2 & 0 & 0 \\ 0 & 0 & \omega_3 & 0 \\ 0 & 0 & 0 & \omega_4 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} \leq \begin{pmatrix} \dot{q}_{0,max} \\ \dot{q}_{1,max} \\ \varepsilon_{1,max} \\ \varepsilon_{2,max} \end{pmatrix}$$

In the first example, two goals are specified but only the first one is selected using the goal's weights. Goal one, located at $q_{des1} = 0.55$, has a weight of $\omega_3 = 0.1$ and the second goal, located at $q_{des2} = 0.75$, has a weight of $\omega_4 = 0.001$, the initial position of the EEF is at $q_{eef} = 0.7$. As

shown in figure 3.11(a), the EEF reaches the first goal (thick dotted line).



(a) First Goal (b) Second goal

Figure 3.11: The selected goal changes depending on the weights. Trajectory (dashed) and velocity of the EEF.

By inverting the weights, $\omega_3 = 0.001$ and $\omega_4 = 0.1$, the active goal changes (figure 3.11 (b)). It can be seen that the velocity of the EEF in figure 3.11 (b) is lower than in (a). This is caused due to the proximity of one of the joints to the joint limit, which was reached shortly after 50 iterations.

It is possible to make the EEF reach a point between both goals, by setting similar or equal weights to both goals (figure 3.12).

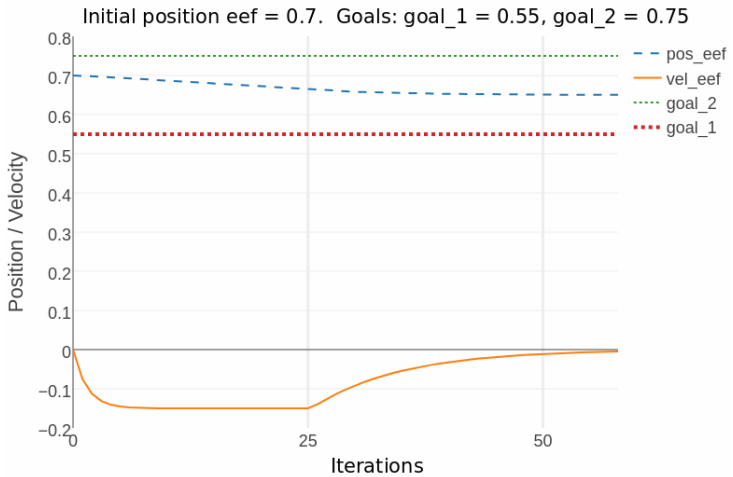


Figure 3.12: Trajectory (dashed) and velocity of the EEF. Same weight in both goals. Each joint has a different velocity and acceleration constraint, as well as different joint limits.

If the weight of both goals is similar, the EEF will stay closer to one goal, without reaching it. Selecting the appropriate weight combination, any point between both goals can be reached.

3.3.5 Dynamic weights

It is possible to automatically select between two goals using dynamic weights, which change with the distance. This controller calculates the weight of each goal in every iteration, with the error to the goals q_{des1} and q_{des2} as:

$$error_1 = (q_{des1} - q_{eef}) * p$$

and

$$error_2 = (q_{des2} - q_{eef}) * p$$

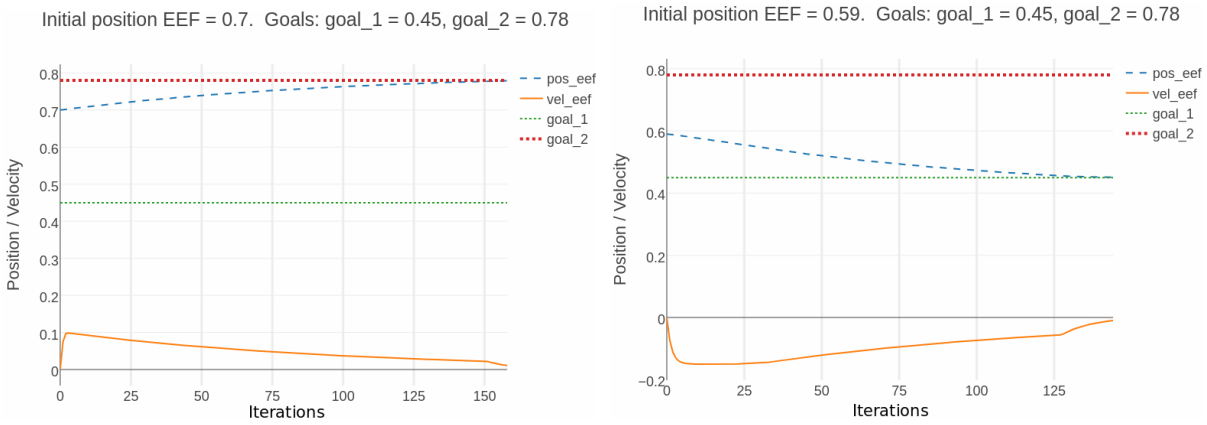
where p is the proportional gain and q_{eef} the current position of the EEF. The weights are calculated with:

$$\omega_3 = \left(\frac{error_2}{error_1 + error_2} \right)^2 \tag{3.20a}$$

$$\omega_4 = \left(\frac{error_1}{error_1 + error_2} \right)^2 \tag{3.20b}$$

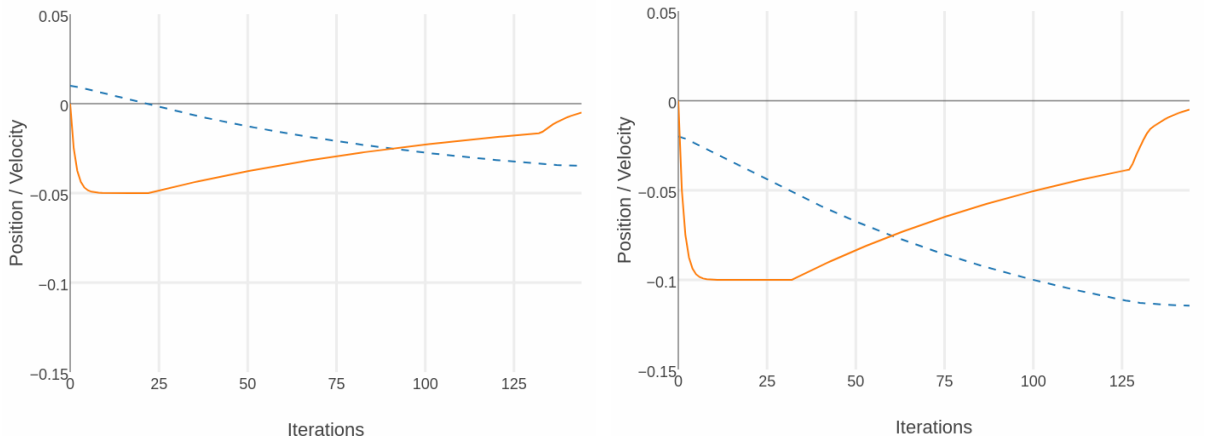
The controller will then select the closest goal, which gets a higher weight.

In this example, the initial position of the EEF is set between both goals. The first goal is set at 0.45 and the second one at 0.78. As shown in figure 3.13, the EEF reaches the closest goal. The only parameter changed between figure 3.13 (a) and (b) is the EEF's initial position.



(a) EEF closer to goal 2 (thick dotted line) (b) EEF closer to goal 1 (thin dotted line)

Figure 3.13: The selected goal changes depending on the initial position of the EEF, weights are calculated on every iteration. Trajectory (dashed) and velocity of the EEF.



(a) Joint q_0 (b) Joint q_1

Figure 3.14: Trajectories (dashed) and velocities of both joints. Each joint has a different velocity and acceleration constraint, as well as different joint limits.

Figure 3.14 shows the trajectories followed by both joints in the second example (3.13 (b)). It can be seen that the velocity profile of both joints are similar. The difference in scale is due to the velocity constraints each joint has.

3.3.6 Three goals for the EEF

A problem seems to appear if dynamic weights are calculated when three or more goals are specified. Taking three goals as an example, if two of them are close together (q_{des2} and q_{des3}), the controller will ignore the third goal (q_{des1}) and reach for a point between the other two goals, since minimizing two errors is better (mathematically speaking) than one.

This problem can be solved by modifying the formula used to calculate the goal's weights, so we can rewrite equation 3.20 as:

$$\omega_3 = \left(\frac{1 - |error_1 - error_{min}|}{error_{max} + error_{min}} \right)^3 \tag{3.21a}$$

$$\omega_4 = \left(\frac{1 - |error_2 - error_{min}|}{error_{max} + error_{min}} \right)^3 \tag{3.21b}$$

$$\omega_5 = \left(\frac{1 - |error_3 - error_{min}|}{error_{max} + error_{min}} \right)^3 \tag{3.21c}$$

where:

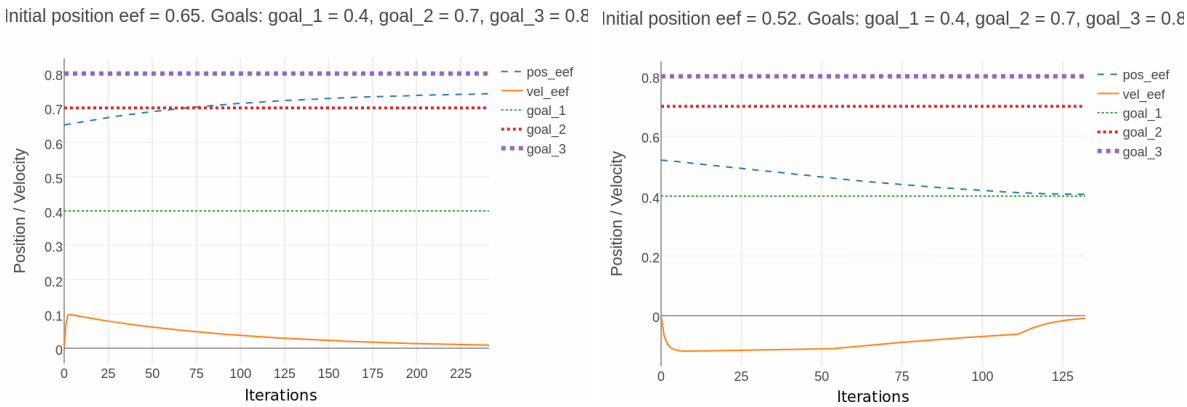
$$error_{min} = min(|error_1|, |error_2|, |error_3|)$$

$$error_{max} = max(|error_1|, |error_2|, |error_3|)$$

The matrix **A** and it's limits as well as the state vector are extended to fit the extra goals:

$$\begin{pmatrix} error_1 \\ error_2 \\ error_3 \\ -q_{0,max} - q_0 \\ -q_{1,max} - q_1 \\ a_{0,min\_lim} \\ a_{1,min\_lim} \end{pmatrix} \leq \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{pmatrix} \leq \begin{pmatrix} error_1 \\ error_2 \\ error_3 \\ q_{0,max} - q_0 \\ q_{1,max} - q_1 \\ a_{0,max\_lim} \\ a_{1,max\_lim} \end{pmatrix}$$

Two examples in figure 3.15 show the behavior of this controller



(a) EEF closer to goal 2 (dotted)

(b) EEF closer to goal 1 (thin dotted)

Figure 3.15: The selected goal changes depending on the initial position of the EEF. Trajectory (dashed) and velocity of the EEF.

The controller does not always reach a goal, as seen in figure 3.15 (a), if two of the given goals are close together, the EEF will stay between both.This can still be implemented as a controller for a robot if the goals are grasping poses of an object, since the gripper will still reach out for the object and in some cases a point between grasping poses can also be used to grasp the object, like when grasping a box.

The code generated for this controllers can be downloaded from Github[5].

3.3.7 Robot controller

A controller with the same structure as the ones shown in the previous subsections was developed for the Boxy robot. Boxy's geometry and kinematics are taken from the URDF file (section 2.6).

The definition of the matrix $\mathbf{A}$ and of the state vector $\vec{s} \in \Re^{nV}$ changes due to the DOF's and

[5]https://github.com/mgvargas/qpOASES

kinematics of the robot, so equation 3.2 changes to:

$$\vec{s} = [\ \dot{q}_0\ \ \dot{q}_1\ ...\dot{q}_C\ \ \varepsilon_0 ... \varepsilon_6]^T \tag{3.22}$$

where $\dot{q}_0$ to $\dot{q}_C$ are the joints velocities, three for the base, one for the torso and seven for the arm, and ε_0 to ε_6 are slack variables, one per DOF. The matrix $\mathbf{A} \in \Re^{nC \times nV}$ is defined using the Jacobian:

$$\mathbf{A} = \begin{bmatrix} \mathbf{J} & \mathbf{I}_{p,p} \\ \mathbf{I}_{m,m} & \mathbf{0}_{m,p} \\ \mathbf{I}_{m,m} & \mathbf{0}_{m,p} \end{bmatrix} \tag{3.23}$$

where $\mathbf{J} \in \Re^{p \times m}$ is the Jacobian, $\mathbf{I}$ is the identity matrix, $\mathbf{0}$ is a zero matrix, p is the number of DOF and m the number of joints to control. The constraint vectors are created using the error, joint limits and acceleration constraints:

$$\vec{lbA} = [\ \vec{error}_{pos},\ \vec{error}_{orient},\ -\vec{q}_{max} - \vec{q},\ \vec{a}_{min\_lim}]^T \tag{3.24a}$$

$$\vec{ubA} = [\ \vec{error}_{pos},\ \vec{error}_{orient},\ \vec{q}_{max} - \vec{q},\ \vec{a}_{max\_lim}]^T \tag{3.24b}$$

where the acceleration limit is calculated using equation 3.16, $\vec{q}_{max}$ is the vector of joint limits and $\vec{q}$ the current position of the joints. The $\vec{error}_{pos}$ is the position error and $\vec{error}_{orient}$ the orientation error.

The position error is calculated considering a proportional gain, to accelerate the convergence of the system:

$$\vec{error}_{pos} = (\vec{q}_{des} - \vec{q}_{eef}) * p$$

where $\vec{q}_{des}$ is the goal position, $\vec{q}_{eef}$ the current position of the EEF and p is the proportional gain.

However, calculating the orientation error cannot be done by simply subtracting goal orientation

Algorithm 2 Orientation Error using Slerp

1: $q_{whole\_error} = q_{goal} * \ q_{eef}^{-1}$;
2: $rot_{vector} = quat\_to\_axisangle(\ q_{whole\_error})$;
3: $rot_{error} = \sqrt{rot_{vector}[0]^2 + rot_{vector}[1]^2 + rot_{vector}[2]^2}$
4: **if** $t = threshold - rot_{error} \geq 0$ **then**
5: $\quad t = 1$;
6: **else**
7: $\quad threshold / rot_{error}$;
8: **end if**
9: $q_{interpolation} = Slerp\ (t; q_{eef}, q_{goal})$;
10: $q_{error} = q_{interpolation} * \ q_{eef}^{-1}$;

from the current one. Several approaches for solving this problem are explained by Siciliano et al. (2008). In this work, the orientation error is calculated using quaternion *spherical linear interpolation* (Slerp). The Slerp gives an interpolation between 0 and 1 from the given quaternions. It was developed by Shoemake (1985):

$$Slerp(t; q_0, q_1) = \frac{q_0 \sin((1-t)\theta) + q_1 \sin(t\theta)}{\sin\theta}$$

for $0 \leq t \leq 1$. Using Slerp, it was possible to calculate an orientation error for the system set as goal for every iteration a small part of the total error. Using the quaternion obtained by the Slerp, the error can be calculated, as explained by Siciliano et al. (2008), as:

$$q_{error} = q_{goal} * q_{current}^{-1}$$

The Algorithm 2 is used to calculate the error and the obtained quaternion is transformed to euler angles and given as $\vec{error}_{orient}$ to the constraint vectors 3.24.

The weight vector $\vec{\omega}$ is calculated on every iteration and used to update matrix $\mathbf{H}$, which is used to calculate the cost in equation 3.6.

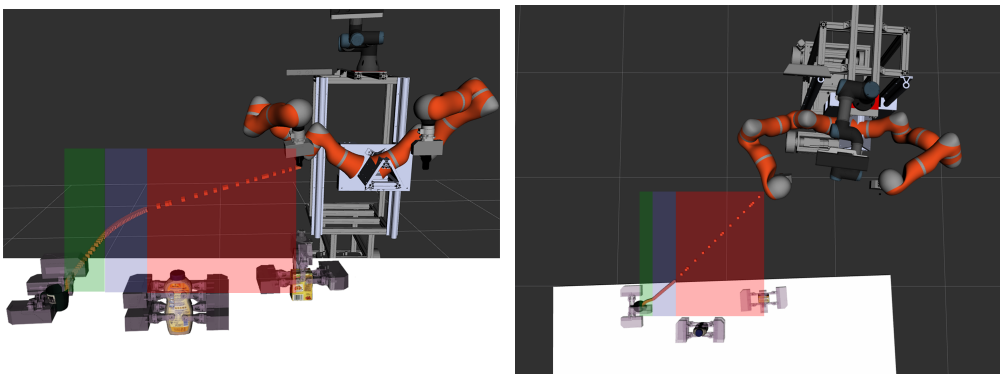
$$\mathbf{H} = diag(\vec{\omega}) \tag{3.25}$$

Algorithm 3 Dynamic weight calculation

if $distance \geq far$ **then**
  $weights_{base} = weight_{active}$;
  $weight_{torso} = weight_{active}$;
  for n in joints **do**
    $weights_{arm}[n] = weight_{inactive}$;
  end for
else if $near \leq pos_{robot} \leq far$ **then**
  for n in joints **do**
    $weights_{arm}[n] = \frac{(weight_{inactive} - weight_{active}) * (distance - near)}{far - near} + weight_{active}$;
  end for
  $weights_{base} = \frac{(weight_{active} - weight_{inactive}) * (distance - near)}{far - near} + weight_{inactive}$;
  $weight_{torso} = weights_{base}$;
else
  $weights_{base} = weight_{inactive}$;
  $weight_{torso} = \frac{(weight_{active} - weight_{inactive}) * (abs(error_{pos}[2]))}{far} + weight_{inactive}$;
  for n in joints **do**
    $weights_{arm}[n] = weight_{active}$;
  end for
end if

The value of the weights depends on the distance between the robot and the goal, the calculations are explained in algorithm 3. Where *distance* is the distance between the EEF and the goal, $weight_{active}$ and $weight_{inactive}$ are constant values, $weight_{active} \ll weight_{inactive}$, far and *near* are threshold distances.

If the robot is too far away from the given goal, the arm joints are disabled and only the base moves. Between a certain distance threshold, both base and arms can move, with the weights dynamically changing in accordance to the distance. If the robot is close to the goal, only the arms and torso are allowed to freely move, the weight of the base is increased so that the controller prefers moving arms and torso to base.



(a) Isometric view (b) Top view

Figure 3.16: Trajectory (dashed) of the EEF. The weights are calculated on every iteration.

Figure 3.16 shows a simulated trajectory followed by the EEF while reaching out for a cup. The right area in figure 3.16 (b) shows the part of the trajectory where mainly the base was moved, the middle area is where arms, torso and base were active and the left area is where mainly the arm and torso were used.

The first goal set to the controller is a pre-grasping pose, located eight centimeters away on the Z-axis from the current grasping pose. Once the EEF is within a certain position and orientation threshold from the pre-grasping pose, it moves towards the final grasping pose. The pre-grasping pose is defined to help avoiding collisions with the object, the EEF can approach the object only when the orientation is already correct and will do so in a way that it will not try to go through the object in order to reach the grasping pose.

3.4 Trajectory Evaluation

When the projection manager system receives a request to grasp an object, it asks the controller for several trajectories. The trajectory evaluation service is in charge of assigning scores to each one on these trajectories and sending the best one back.

First, each trajectory is sent to the *collision detection* node, if a collision is found, the trajectory gets discarded, if not, the minimum distance between the robot and an object is obtained. The remaining trajectories are then evaluated.

For the evaluation, different metrics are considered

- Length of the generated path
- Smoothness: No abrupt changes in velocity and acceleration
- Convergence error: How far was the arm from the desired position
- Manipulability of the EEF after reaching the goal
- Distance to collision: Minimum distance from the arm to any object in the scene

The manipulability is calculated as proposed by Yoshikawa (1985), who defines it as a quality measure for redundant manipulators, it describes the distance to singular configuration of the robot.

$$manipulability = \sqrt{det(\mathbf{J}\mathbf{J}^T)}$$

where $\mathbf{J}$ is the Jacobian of the kinematic chain used to perform the trajectory. This approach was also used by Vahrenkamp et al. (2012), they combine the manipulability with a penalization term that considers the distance to the joint limits to measure the robot's ability to maneuver in the workspace.

After simulation several experiments, a maximum value for acceleration and velocity change was defined, as well as for manipulability. The maximum length of the trajectory was obtained based on the robot arm workspace. Using these values, a score for each trajectory can be

calculated:

$$\begin{aligned} score = w_l \frac{traj\_length}{max\_length} + w_v \frac{vel\_change}{max\_vel\_change} + w_a \frac{acc\_change}{max\_acc\_change} \\ + w_c \frac{collision\_distance}{min\_col\_distance} + w_e \frac{position\_error}{max\_position\_error} \\ + w_m(max\_manip - manipulability) \end{aligned} \tag{3.26}$$

where w_l, w_v, w_a, w_c, w_e w_m are the weights for length, velocity change, acceleration change, distance to collision, convergence error and manipulability, respectively.

The trajectory with the lowest score is selected and sent back to the *projection_manager*, which sends it as a result to a PI controller. This controller sends the required commands to the robot to execute the trajectory.

3.5 Collision Checking

Since the motion controller is not able to detect collision objects in the environment, it is important to evaluate if the obtained trajectory would generate a collision if executed. To do so, the collision detection functionality of *Moveit! Planning Scene* (section 2.5.6) was used. All objects detected by perception, as well as the collision model of the robot were loaded in a planning scene. The robot's URDF (section 2.6) is also added to the planning scene, this allows to evaluate if the robot is in a valid state (feasibility) and to calculate the state of the robot given the joint values.

The object the robot will grasp is removed from the collision objects, since the grasping would be considered as a collision. This has the disadvantage that the robot's EEF might hit the object while reaching for the grasping pose and the collision detection would not perceive it. In order to reduce the possibility of this collision, a pre-grasping pose is set as initial goal for the EEF,

this pose is located 3 cm away from the object. Only after the EEF has reached the pre-grasing pose, it approaches to the object.

For every step in the trajectory, the new joint values are loaded in the robot model. The planning scene evaluates if the robot is in a valid state and if there is any collision, both self-collisions and collisions with the environment are considered. If no collision was found, the minimum distance to an object is stored. The evaluation stops if a collision is found and the trajectory is discarded.

3.6 PI Controller

The Boxy robot moves by receiving a sequence of velocity commands for the joints. This means: if the robot receives the command "move the base at $0.5\frac{m}{sec}$" during 2 seconds, it will execute this movement for two seconds. Therefore it is not possible to send a whole trajectory at once and expect the robot to move, since the trajectory is a sequence of desired positions and velocities.

The proportional and integral controller (PI controller) reads the trajectory step by step and sends the specified position for each joint. It reads the time stamps of the trajectory to find out where the robot should be after a certain time interval. It uses a proportional gain and an integral gain to compensate for the robot's inertia.

The *naive_kinematic_simulator* (section 2.5.5) used to generate the trajectories assumes that the robot will instantaneously execute the received commands and reached the desired speed. In reality, the robot needs some time to reach a certain speed due to inertia. The motion controller tries to consider this delay by setting acceleration limits while calculating the trajectories, but it

stills needs some help from the PI controller to make the robot follow the desired trajectory.

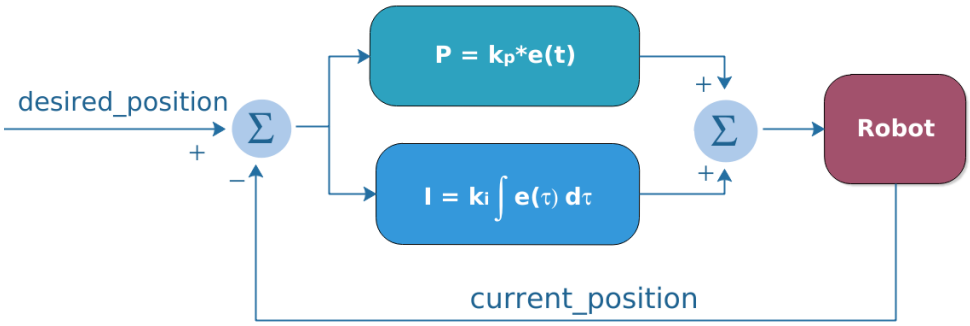


Figure 3.17: Diagram of PI Controller

Figure 3.17 shows the diagram of the used PI controller. It was tuned to work with the Boxy robot first by tunning it with the *naive_kinematic_simulator* and afterwards with the real robot.

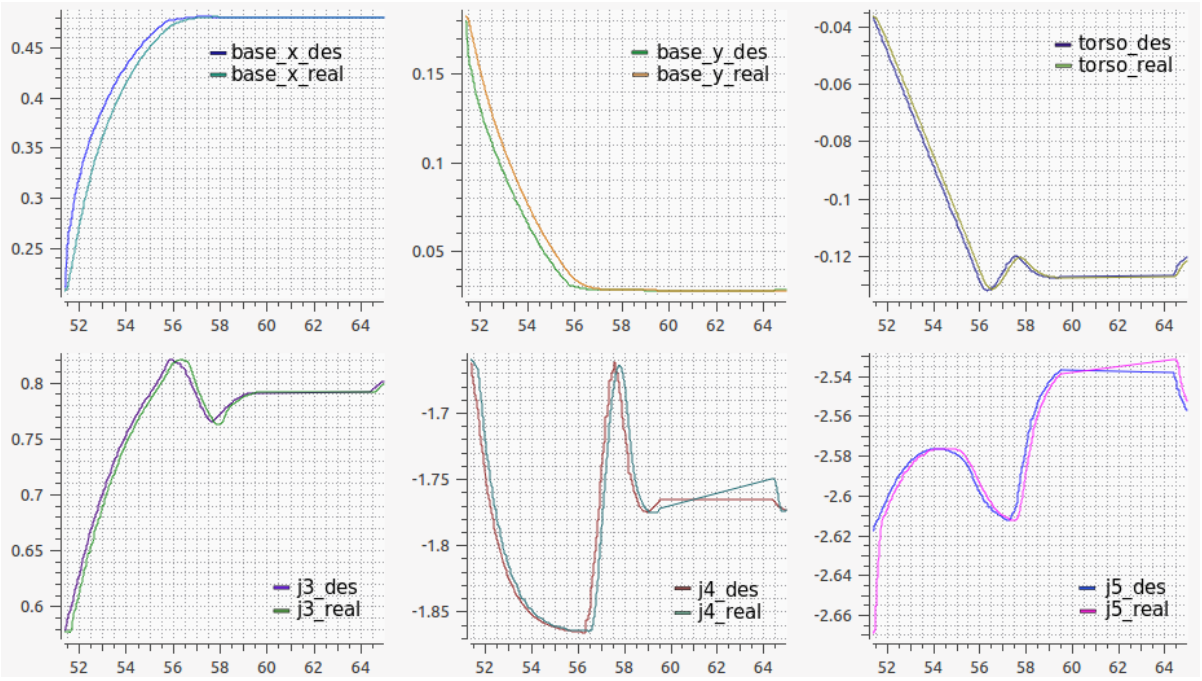


Figure 3.18: Comparison between generated and executed trajectories. X axis - seconds, Y axis - joint values

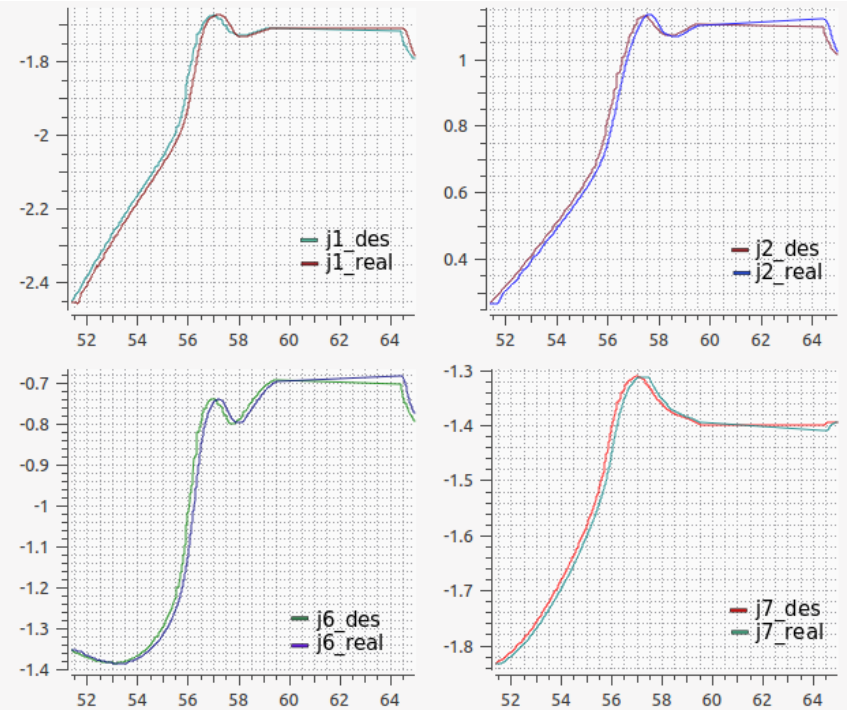


Figure 3.19: Comparison between generated and executed trajectories. X axis - seconds, Y axis - joint values

Figures 3.18 and 3.19 show a comparison between the trajectory generated by the motion controller and the one executed by the robot. The first three plots if figure 3.18 show the behavior of the base and torso. The remaining plots show the seven joints of the arm used to grasp the object. As it can be seen, the robot follows accurately the planned trajectory.

CHAPTER 4

Evaluation and Results

The system created during this thesis receives a command with the name of the object the robot should grasp. After it detects the object, the projection manager (section 3.1) requests several trajectories to the motion controller. These reach different grasping poses using both arms (one at a time).

In order to measure the quality of the trajectories obtained by the controller and decide if the grasping action was successful, these trajectories must be evaluated (section 3.4).

The metrics considered for each trajectory are:

- Length
- Smoothness
- Convergence error
- Manipulability of the arm after reaching the goal
- Distance to collision

All trajectories that generate a collision or that do not reach a certain threshold distance around the grasping pose are discarded. The remaining trajectories are scored based on these metrics

and the best one is sent to the robot.

In the simulated experiments, grasping is considered successful if the robot's EEF reached one of the defined grasping poses without collisions. For the experiments with the real robot, grasping was considered successful if the robot grasped and lifted the object.

4.1 Experimental Setup

The first experiments were made in simulation and visualized using RVIZ (section 2.5.2). Several initial configurations were tested. After achieving suitable results, experiments were done using the real robot.

4.1.1 Scenario 1: Simulation

The scenario simulates three objects detected on top of a table: a cup, a tomato sauce package (knorr tomate) and a bottle of pancake mix (mondamin). A model of the robot was loaded in this environment in front of the table (figure 4.1). The table model and the general position of the robot with respect to it simulate the environment of the IAI's lab, the place where the experiments with the robot were done.

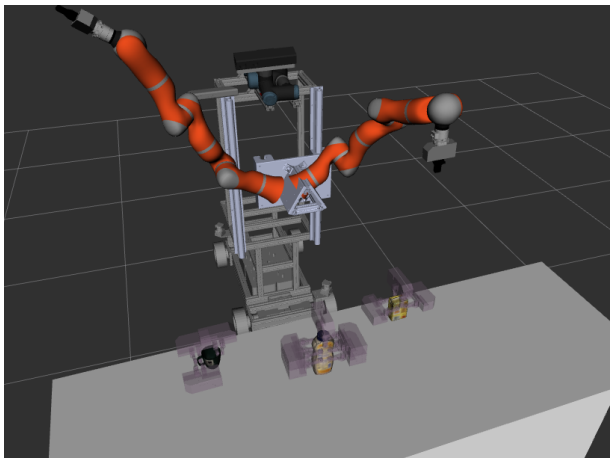


Figure 4.1: Simulation Environment

Ten different initial configurations for the robot were defined. For each configuration, the robot had to grasp the three objects shown in figure 4.1. The system was told to calculate five trajectories each time it received the request to grasp an object, and select the one with the best score.

For some of the initial configurations, the robot was positioned far away from the table, so that the objects were out of reach. This was done in order to test the base's movement. In other cases the robot was placed closer to the table, so that the arms and torso started moving from the beginning of the trajectory.

The experiments were repeated with three different poses for each object, giving a total of 90 grasping simulations.

4.1.2 Scenario 2: Testing with the robot

The second group of experiments was executed on the real robot. The robot was placed in front of a table where some objects from the database were placed. In this case, the trajectory evaluation included the execution time (the time the robot required to execute the given trajectory).

For this tests with the Boxy robot, grasping was considered successful if the robot could grasp and lift the object for at least 3 seconds. Twelve experiments were done, with three objects in four different positions. The initial pose of the robot was set per hand before each experiment, so there are variations between each pose.

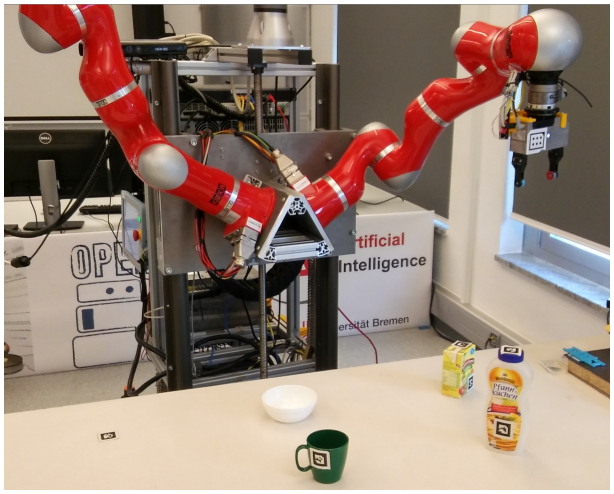


Figure 4.2: Location array of the objects during the experiments

Figure 4.2 shows the four locations where each object was placed during the grasping experiments. During some experiments, objects were placed close to the object the robot grasped, to test if grasping in a packed environment could be achieved.

4.2 Experimental Results

Figure 4.3 shows the results obtained by one iteration of the motion controller. In this case, the right arm of the robot has to grasp a cup located on a table out of reach for the robot.

Analyzing the first two plots of figure 4.3, it can be seen how the weights dynamically change with the distance. These are the weights used in equation 3.25 to update the **H** matrix, therefore, to update the cost function (equation 3.6) used by the controller.

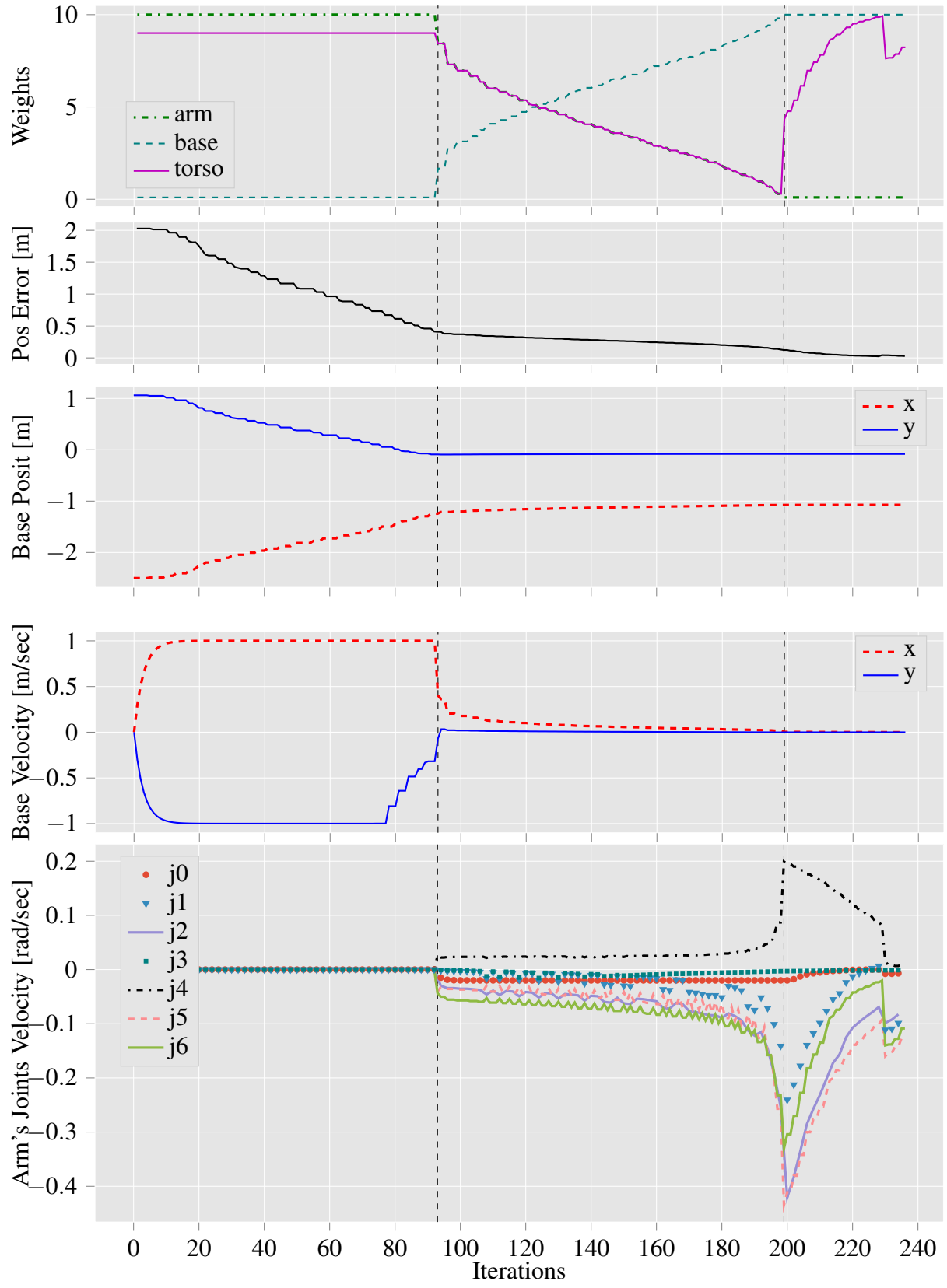


Figure 4.3: Behavior during one simulated trajectory. Iteration 0 to ≈ 100: base active. Iteration ≈ 100 to ≈ 200: Whole robot active. Iteration ≈ 200 on: Arm active, torso partially active.

During the first hundred iterations, the robot is too far away from the table, so it will move only the base. Between a certain threshold, the base's weights starts increasing and the arm's and torso's weights decrease, the robot starts to use more the arm and torso and less the base. When Boxy approaches the table, the base's weight is high, so the movements are executed mainly with the arm.

At the beginning of the trajectory, the error decreases fast, because the robot is just approaching to the table moving the base. When only the arm and torso are active, the error converges slowly, since the controller must now consider not only position, but also orientation.

4.2.1 Scenario 1: Simulation

Table 4.1 shows the scores obtained by each trajectory of the simulation experiments described in section 4.1.1. Trajectories that did not reached the goal in a given time were considered as failed and were not evaluated. The scores of the trajectories with a collision are also not displayed since these trajectories were discarded. The evaluation was done as explained in section 3.4.

To make the table easier to read and compare values, the failed trajectories are written below the successful ones.

Scenario 1: Trajectory simulation							
Object	Length	Smoothness		Dist. to collision	Manipulability	Posit. Error	Score
		Velocity	Acceleration				
Initial robot configuration 0							
cup	180.73	0.287	0.561	4.64	0.106	0.71	2.509
	180.77	0.301	0.584	4.65	0.106	0.73	2.551
	182.05	0.209	0.408	7.64	0.122	0.62	2.617
	181.88	0.185	0.363	7.79	0.122	0.66	2.682
	Trajectory failed						

Scenario 1: Trajectory simulation							
Object	Length	Smoothness		Dist. to collision	Manipulability	Posit. Error	Score
		Velocity	Acceleration				
mondamin	178.87	0.258	0.5	2.34	0.109	0.65	2.17
	179.14	0.258	0.487	2.37	0.109	0.67	2.192
	193.8	0.276	0.539	8.67	0.106	0.6	2.772
	193.87	0.272	0.532	8.73	0.073	0.62	2.865
	197.01	0.263	0.49	6.02	0.074	0.67	2.675
knorr tomate	158.88	0.192	0.35	3.68	0.109	0.71	2.331
	158.91	0.183	0.326	3.68	0.109	0.73	2.354
	154.08	0.226	0.442	10.39	0.105	0.78	3.112
	160.65	0.268	0.499	5.6	0.122	0.73	2.552
	201.92	0.305	0.542	1.72	0.106	0.64	2.155
Initial robot configuration 1							
cup	180.77	0.308	0.6	4.64	0.106	0.72	2.528
	194.31	0.382	0.71	5.82	0.106	0.75	2.75
	201.45	0.198	0.376	7.13	0.089	0.82	3.0
	201.44	0.205	0.399	7.06	0.089	0.83	3.008
	Trajectory failed						
knorr tomate	187.69	0.235	0.448	10.53	0.088	0.58	2.934
	187.75	0.235	0.452	10.52	0.088	0.59	2.944
	Trajectory failed						
	Trajectory failed						
	Trajectory failed						
Initial robot configuration 2							
cup	201.45	0.184	0.36	7.07	0.069	0.82	3.025
	Collision Found						
	Collision Found						
	263.84	0.208	0.406	4.35	0.121	0.8	2.782
	Trajectory failed						
mondamin	Collision Found						
	Collision Found						
	Trajectory failed						
	Trajectory failed						
	Trajectory failed						
knorr tomate	266.42	0.356	0.692	6.73	0.121	0.66	2.846
	265.97	0.342	0.662	6.75	0.121	0.67	2.851
	262.83	0.204	0.371	8.13	0.054	0.77	3.235
	Collision Found						
	Trajectory failed						
Initial robot configuration 3							
cup	Collision Found						
	199.8	0.228	0.442	8.03	0.104	0.69	2.853

Scenario 1: Trajectory simulation							
Object	Length	Smoothness		Dist. to collision	Manipulability	Posit. Error	Score
		Velocity	Acceleration				
	197.88	0.328	0.635	5.75	0.118	0.77	2.742
	222.89	0.279	0.482	8.38	0.13	0.8	3.072
	Trajectory failed						
mondamin	216.69	0.26	0.483	2.0	0.099	0.6	2.167
	216.74	0.267	0.506	1.93	0.099	0.59	2.145
	213.87	0.276	0.521	2.3	0.042	0.62	2.341
	Trajectory failed						
	Trajectory failed						
knorr tomate	225.0	0.383	0.746	5.4	0.13	0.9	2.976
	225.03	0.422	0.792	5.44	0.13	0.91	2.994
	217.57	0.256	0.492	2.79	0.105	0.58	2.199
	217.58	0.243	0.433	2.79	0.044	0.6	2.351
	225.05	0.391	0.774	5.39	0.13	0.9	2.981
Initial robot configuration 4							
cup	199.75	0.236	0.456	8.19	0.105	0.73	2.939
	305.56	0.295	0.572	8.53	0.105	0.64	3.095
	354.83	0.238	0.446	7.48	0.098	0.83	3.428
	Trajectory failed						
	Trajectory failed						
mondamin	323.73	0.23	0.425	3.52	0.103	0.83	2.931
	323.7	0.234	0.426	3.58	0.103	0.83	2.936
	318.77	0.227	0.427	6.64	0.117	0.82	3.191
	Trajectory failed						
	Trajectory failed						
knorr tomate	314.85	0.253	0.48	10.52	0.102	0.74	3.473
	314.72	0.258	0.502	10.5	0.102	0.73	3.461
	324.16	0.232	0.446	2.2	0.109	0.8	2.75
	Trajectory failed						
	Trajectory failed						
Initial robot configuration 5							
cup	305.63	0.3	0.591	8.45	0.086	0.62	3.094
	133.87	0.394	0.781	8.47	0.086	0.67	2.784
	153.91	0.323	0.605	2.73	0.097	0.65	2.185
	Trajectory failed						
	Trajectory failed						
mondamin	142.06	0.165	0.316	2.74	0.023	0.62	2.207
	142.04	0.166	0.296	2.73	0.023	0.62	2.215
	Trajectory failed						
	Trajectory failed						
	Trajectory failed						

Scenario 1: Trajectory simulation							
Object	Length	Smoothness		Dist. to collision	Manipulability	Posit. Error	Score
		Velocity	Acceleration				
knorr tomate	143.23	0.161	0.278	0.5	0.105	0.81	2.125
	143.43	0.171	0.319	0.42	0.105	0.66	1.889
	145.13	0.198	0.37	10.03	0.031	0.84	3.296
	193.41	0.746	1.373	7.58	0.092	0.89	3.249
	Trajectory failed						
Initial robot configuration 6							
cup	133.75	0.373	0.738	8.49	0.105	0.66	2.721
	Trajectory failed						
	Trajectory failed						
	Trajectory failed						
	Trajectory failed						
mondamin	158.7	0.205	0.355	17.27	0.101	0.65	3.6
	158.82	0.208	0.348	17.21	0.101	0.68	3.642
	Trajectory failed						
	Trajectory failed						
	Trajectory failed						
knorr tomate	162.85	0.13	0.242	9.88	0.092	0.73	3.009
	162.85	0.127	0.239	9.91	0.092	0.74	3.034
	172.16	0.332	0.565	6.19	0.039	0.66	2.7
	Trajectory failed						
	Trajectory failed						
Initial robot configuration 7							
cup	118.67	0.231	0.43	4.37	0.126	0.72	2.291
	118.74	0.223	0.432	4.41	0.126	0.73	2.307
	119.27	0.244	0.482	6.59	0.13	0.77	2.584
	Trajectory failed						
	Trajectory failed						
mondamin	139.97	0.318	0.634	2.76	0.13	0.82	2.351
	140.14	0.322	0.629	2.79	0.13	0.82	2.363
	123.96	0.162	0.323	6.39	0.108	0.68	2.46
	123.97	0.168	0.326	6.38	0.107	0.65	2.408
	123.02	0.175	0.344	8.81	0.107	0.68	2.697
knorr tomate	131.93	0.289	0.561	2.91	0.13	0.75	2.224
	132.02	0.394	0.763	2.91	0.13	0.72	2.213
	120.61	0.213	0.415	7.57	0.103	0.96	3.038
	Collision Found						
	Trajectory failed						
Initial robot configuration 8							
cup	205.2	0.387	0.76	2.63	0.0	0.82	2.782
	205.33	0.38	0.741	2.7	0.0	0.82	2.792

Scenario 1: Trajectory simulation							
Object	Length	Smoothness		Dist. to collision	Manipulability	Posit. Error	Score
		Velocity	Acceleration				
	206.7	0.247	0.475	8.62	0.128	1.02	3.411
	206.89	0.246	0.47	8.67	0.128	0.94	3.289
	Trajectory failed						
mondamin	205.98	0.409	0.811	7.34	0.076	1.0	3.395
	205.79	0.411	0.797	7.37	0.076	0.98	3.371
	205.75	0.405	0.807	7.35	0.028	0.99	3.478
	Trajectory failed						
	Trajectory failed						
knorr tomate	218.88	0.212	0.413	10.85	0.056	1.02	3.803
	218.75	0.214	0.413	10.83	0.056	1.05	3.845
	260.76	0.357	0.62	6.26	0.104	0.7	2.875
	Trajectory failed						
	Trajectory failed						

Table 4.1: Simulation Experiment: Results of trajectory evaluation for ten initial configurations where three objects were grasped. Only the first 25 experiments are shown

For each object, the trajectory with lower score was selected as the one to be sent to Boxy. The implications of this table are discussed in the next chapter.

Scenario 1: Summary						
Grasping events	Num. Trajectories	Trajectories			Failed Grasping	Successful Grasping
		Successful	Failed	Collision		
90	450	247	170	33	10	80

Table 4.2: Simulation Experiment: Summary of trajectory evaluation for ten initial configurations where three objects were grasped with three positions per object

Table 4.2 shows a summary of the experiment results, classifying the trajectories in successful, failed or collision detected. From the 90 grasping actions simulated, 88.9% were successful.

4.2.2 Scenario 2: Testing with the robot

After running the first experiments with the robot, I realized that some parameters had to be changed in order to successfully grasp objects:

- The pre-grasping pose has to be further away from the actual grasping pose to avoid collision between the gripper and the object. The distance changed from three to eight centimeters.
- The velocity limit of the base specified in the URDF is $1\frac{m}{s}$, this is too high for the real robot. It was reduced to $0.3\frac{m}{s}$, because higher velocities caused the robot's structure to oscillate.
- The error margin of the pre-gasping pose also had to be reduced to avoid colliding with the object while approaching to the grasping pose.

One problem I noticed with the perception system is that the orientation of the fiducial markers used to detect the objects is not always correct, this can lead to a fail grasping. Usually rotating the objects corrected the problem.

Finally, I successfully ran experiments with Boxy. However, the process was slow due to the time required for collision detection, approximately 16 seconds per trajectory. Trying to reduce this time, I simplified the collision models of the objects. This reduced the average collision detection time to 9.2 seconds.

Table 4.3 shows the results of the experiments I did on the robot after doing the changes mentioned above. It shows the time required by the robot to execute the trajectory, the position error with respect to the planned trajectory and the outcome of the grasping. The position error was calculated with the joint states measured by the robot, it was not taking into account possible errors in the base odometry.

Scenario 1: Experiments with the Boxy Robot				
Object	Pose Grasped	Duration [sec]	Error wrt planned [cm]	Result (Object state)
mondamin	GP 5	6	1.1	Collision w/ gripper
	GP 1	7	0.45	Grasped
	GP 5	11	0.7	Grasped
	GP 5	9	0.8	Grasped
knorr tomate	GP 3	7	1.4	Grasped
	GP 3	8	0.6	Dropped
	GP 3	10	0.2	Grasped
	GP 3	7	1.2	Grasped
cup	GP 1	13	1.03	Collision w/ gripper
	GP 3	12	0.8	Grasped
	GP 3	22	1.1	Did not reached
	GP 3	24	0.5	Did not reached

Table 4.3: Experiment Results: Grasping actions performed with the Boxy robot

There were problems with the detection of the cup, due the location of the markers. One of the markers was often detected closer than it really was, so the gripped did not reached the object.

In 10 out of the 12 experiments, the planner decided to grasp the pose located on top of the object, since, for most common configurations or the arm, is the easiest to reach.

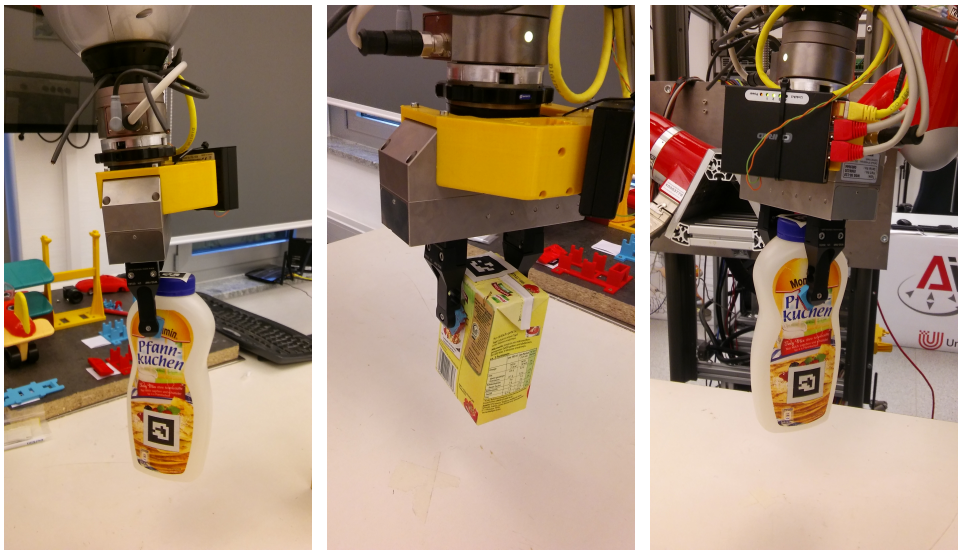


Figure 4.4: Grasping examples

Three examples of successful grasping are shown in figure 4.4.

Figure 4.5 shows two scenarios where a successful grasping was made in a situation where the object to grasped was surrounded by other objects.

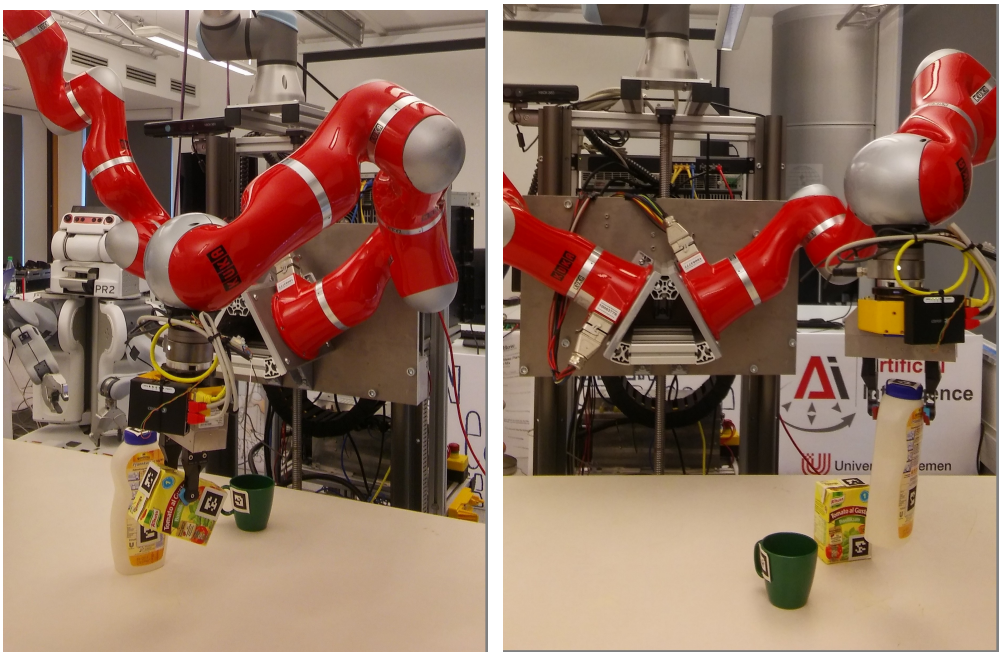


Figure 4.5: Grasping in a crowded scenario

During the tests, only one of the robots could be used due to a internal problem of the right arm, so no comparison between the results of each arm could be done. However, after tuning the system, most of the grasping actions were successful. No trajectory executed generated a collision, even when objects were near the goal position.

CHAPTER 5

Discussion & Conclusion

The aim of this work is to develop a system with a motion controller for the Boxy robot that generates trajectories that allow the robot to successfully grasp a specified object. The system must be able to decide on its own how to grasp the object. Velocity and acceleration constraints are considered by the motion controller while generating the trajectories.

When the system receives a request of generating a trajectory to a given object, it first access the object database (section 3.2.1) to retrieve the grasping poses of the object. Then, based on the distance of each grasping pose to the robot and the manipulability of both arms, it selects a grasping pose and generates several trajectories. Afterwards, it repeats this process with a couple of the remaining grasping poses. Finally, the system must decide which one of the obtained trajectories is better and sends it to the robot.

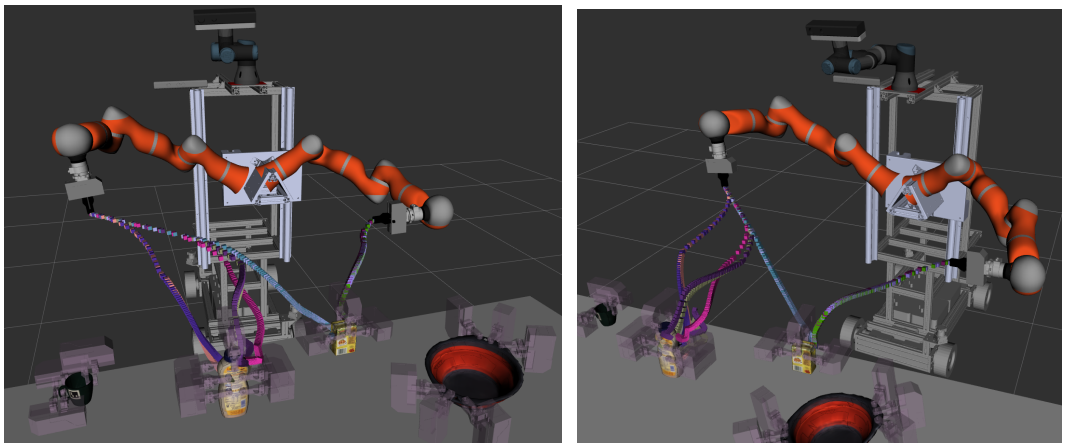
The system is comprised by perception, an object database, a motion controller for the Boxy robot, a trajectory evaluation service, collision detection, a PI controller and a projection manager that coordinates the other elements and is in charge of decision making.

From the table shown in section 4.2.1, it can be seen that the initial configuration of the robot has an important influence in the success rate of the system. The motion controller tries minimize the position error of the end effector, it searches the shortest path to reach the goal. If one

or more of the arm's joints reaches joint limits in the middle of the trajectory generation, the controller won't try to move the joint in the other direction, since this would mean increasing the position and orientation error. This leads to a failed trajectory, since the controller is not able to find another solution.

However, in 24 of the 25 experiments shown in table 4.1, the controller was able to compute at least one successful trajectory. Figure 5.1 shows the result of sending a grasping command for the pancake mix bottle and the tomato sauce package. For the bottle, trajectories to three different grasping poses were planned with the right arm. For the tomato sauce, both arms reached a different grasping pose, the closest one for each.

It is worth mentioning that, for testing purposes, the robot was asked to generate a trajectory to different grasping poses of the object, including the ones in the back of the object. About 50% of the failed trajectories were due to poses the robot could not possibly reach. When using the system on the real robot, the controller generated trajectories only for reasonable grasping poses.



(a) (b)

Figure 5.1: Result of simulated grasping of two objects. Five trajectories per object were calculated.

Calculating each trajectory takes, in average, 5.9 seconds. This is the time the robot needs to execute the trajectory. On each iteration of the calculation, the controller sends a velocity command to the *naive_kinematic_simulator*, the simulator executes this command for a small amount of time (0.01 seconds) and sends the new state back to the controller. The controller repeats this process until the robot reaches the goal position. It needs to know how long the robot must execute each command to reach the goal. This means that generating the trajectory takes as long as executing it.

However, the trajectory evaluation considerably slows down the process due to the collision detection. Checking for collisions in one trajectory lasts 9.2 seconds in average. This process is slow due to the amount of points in the 3D space that must be evaluated.

As a contribution in future work, lowering the time required for collision checking could be achieved by the development and implementation of improved collision models. Another approach could be to preload all models that might be used and then remove the ones not required before evaluating. One possible optimization is to make an interaction between generating the trajectory and evaluating it, in order to discard the trajectory as soon as a collision is found.

The developed system was able to successfully grasp objects in table top scenarios, even when surrounded by other objects. as shown in figure 5.2. This implies that the collision detection of the system allows grasping in complex scenarios.

Figure 5.2: Grasping in a crowded scenario

Some remarks I want to point out after finishing this thesis are:

- No other system with a whole body optimization-based controller that included arms, torso and base for grasping scenarios was found.
- Having a system with predefined grasping poses for given objects simplifies the planning of more complex grasping sequences.
- Integrating the base in the robot's motion controller increases the working space of the robot and speeds un trajectories where the objects are located far away from the end effector
- Implementation of collision avoidance makes the system safer to use in environments where several objects are found.
- Fiducial markers provide a simple and reliable way of locating and identifying objects.

The results of this thesis can help researchers improve the autonomy of robots, by being able to grasp an object without having to give a specific pose as a command. This could allow a greater flexibility of the overall system and shorter the time required to plan new experiments where robots have to interact with objects.

APPENDIX A

Appendix

A.1 Github Repositories

This project was developed on a computer running Ubuntu 16.04, no special hardware is required to execute the generated code.

However, the user needs to have some software and packages installed:

- **ROS:** With a configured workspace. The ROS distro used was *kinetic*. Instalation instructions can be found in ROS.org[1]
- **MoveIt!:** Used here the ROS kinetic release[2] for collision detection. Requires previous installation of ROS

Here you can find all the Github repositories with the code required to run this project.

- Code generated during this project:

  `https://github.com/mgvargas/iai_markers_tracking`

  `https://github.com/mgvargas/iai_trajectory_generation_boxy`

  `https://github.com/mgvargas/qpOASES`
- Boxy's model and description:

  `https://github.com/code-iai/iai_robots`

[1] http://wiki.ros.org/ROS/Installation
[2] http://moveit.ros.org/install/

- Robot's kinematic simulator:

  `https://github.com/code-iai/iai_naive_kinematics_sim`

In the Readme file of the `iai_trajectory_generation_boxy` package, you can find the instructions to install all the required repositories.

A.2 Code Execution

The `projection_system.launch` is the main launch file, contained in the `iai_trajectory_generation_boxy` package. Here, you can configure if the system will be executed on the robot or just as simulation. When the `sim` argument is `false` and the `boxy` is `true`, the system is configured to run on the Boxy robot receiving images from the Microsoft Kinect 2.

To configure the topics where the projection system subscribes to, you must change the corresponding parameters in the `markers_detection.launch` file of the `iai_markers_tracking` package.

In order to send a command to the robot, you must send a request to the `ProjectedGrasping` action with the name of the object to grasp. This can be done simply by publishing the command to a ROS topic:

```
rostopic pub /projected_grasping_server/goal iai_trajectory_generation_boxy
    /ProjectedGraspingActionGoal "header:
seq: 0
stamp:
secs: 0
nsecs: 0
frame_id: ''
goal_id:
stamp:
secs: 0
nsecs: 0
id: 'SOME_ID'
goal:
```

```
13 object: 'OBJECT_NAME'"
```

The object must be already in the database for the robot to detect it and have the possible grasping poses available.

New objects can be added any time to the `database.yaml` file of the `iai_markers_tracking` package following the same syntax of the other objects, explained in section 3.2.1.

The file `send_commands_to_boxy.py` contains the PI controller in charged of sending the obtained trajectory to the robot, it also closes the gripper and lifts the object once the goal is reached.

References

Chitta Sachin, Sucan Ioan, Cousins Steve. ROS Topics, MoveIt! // IEEE Robotics and Automation magazine. 2012. 18–19.

Craig John J. Introduction to Robotics; Mechanics and Control. 2005. Third.

Dmitry Berenson Siddhartha Srinivasa, Kuffner James. Task Space Regions: A framework for pose-constrained manipulation planning // The International Journal of Robotics Research. 2011. 32. 1435–1460.

Fang Zhou, Bartels Georg, Beetz Michael. Learning Models for Constraint-based Motion Parameterization from Interactive Physics-based Simulation // International Conference on Intelligent Robots and Systems (IROS). 2016. 4.

Ferreau H. J., Bock H. G., Diehl M. An online active set strategy to overcome the limitations of explicit MPC // International Journal of Robust and Nonlinear Control. 2008. 18, 8. 816–830.

Ferreau H.J., Kirches C., Potschka A., Bock H.G., Diehl M. qpOASES: A parametric active-set algorithm for quadratic programming // Mathematical Programming Computation. 2014. 6, 4. 327–363.

Han Jiawei, Zheng Yu, Zhou Xiaofang. Computing with Spatial Trajectories. 2011.

Hoppe Prof. Dr. Ronald H.W. Optimization I. 2006. 56–70.

Kröger Torsten. Opening the Door to New Sensor-Based Robot Applications. The Reflexxes Motion Libraries // Robotics and Automation ICRA. 2011.

Kröger Torsten, Wahl Friedrich M. Online Trajectory Generation: Basic Concepts for Instantaneous Reactions to Unforeseen Events // Transactions on Robotics. 2010. 26. 94–111.

LaValle Steven M. Planning Algorithms. Cambridge, U.K.: Cambridge University Press, 2006. Available at http://planning.cs.uiuc.edu/.

Nikolaou Michael. Model Predictive Controllers: A Critical Synthesis of Theory and Industrial Needs // Advances in Chemical Engineering. 2001. 26. 131–204.

Russell R. A. Robot Tactile Sensing. 1990.

Shoemake Ken. Animating Rotation with Quaternion Curves // SIGGRAPH Comput. Graph. VII 1985. 19, 3. 245–254.

Siciliano Bruno, Sciavicco Lorenzo, Villani Luigi, Oriolo Giuseppe. Robotics - Modelling, Planning and Control. 2008. 2nd Printing. (Advanced Textbooks in Control and Signal Processing).

Siliciano Bruno, Khatib Oussama. Springer Handbook of Robotics. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

Vahrenkamp Nikolaus, Asfour Tamim, Metta Giorgio, Sandini Giulio, Dillmann Rüdiger. Manipulability analysis // IEEE-RAS International Conference on Humanoid Robots (Humanoids). XII 2012. 12.

Wallén Johanna. The history of the industrial robot // Technical report from Automatic Control at Linköpings universitet. 2008.

Yoshikawa Tsuneo. Manipulability of Robotic Mechanisms // The International Journal of Robotics Research. 1985. 4, 2. 3–9.

Zhang Zhijun, Zhang Yunong. Acceleration-Level Cyclic-Motion Generation of Constrained Redundant Robots Tracking Different Paths // Transactions on Systems, Man, and Cybernetics. 2012. 42. 1257–1269.