Universität Bremen*

Artificial Intelligence

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 2, 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 point for robotics. Being able to parametrize the movements a robot execute 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 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 2.4.3), 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]. Robots replaced 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 as 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 and execute a trajectory that moves the end effector (EEF) from its current location to the selected grasping pose. This includes trajectory generation and motion control.
- Evaluate if the obtained trajectory will generate a collision with the robot itself or the environment. Discard the trajectories where collisions were detected.

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.

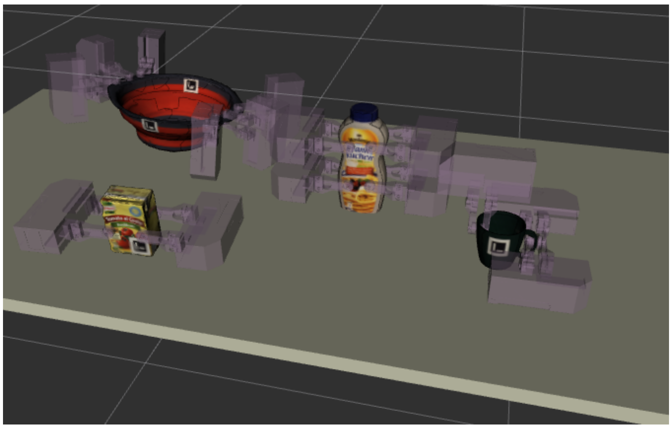


Figure 1.1: Example of objects with predefined grasping poses

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, a optimization controller (section 3.3) will generate several possible trajectories, evaluate them and send the best one to the Boxy robot (figure 1.2).

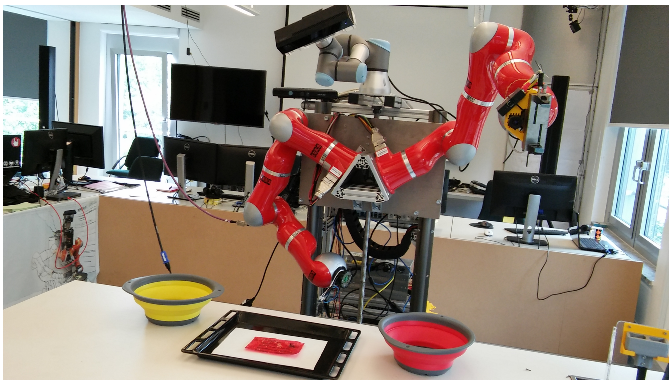


Figure 1.2: The Boxy robot

As result, it is expected a system with a parametrized motion controller that generates a smooth trajectory for Boxy by limiting the velocity and acceleration of each joint. The system should be able to decide which grasping pose (GP) of a selected object can be better grasped given the current system conditions.

is it answering the questions it should?

1.2 Contributions

Data, software, algorithms, designs, etc... made by me If the problem was solved, how was that done?

write

The main contribution was the development of a projection system for trajectory planning (section 3.1). This system uses a naive kinematic simulator 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 2.4.5). 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.

During the development of this work, an object database was created, with 3D models of several objects and predefined grasping poses for each object.

In order to detect the objects located in front of the robot, a perception system that used fiducial markers was developed as side product.

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

1.3 Structure of the thesis

The first chapter explains the aim of this work, as well as the outcomes of it...........

write

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 and Khatib [2008], *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 2 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.

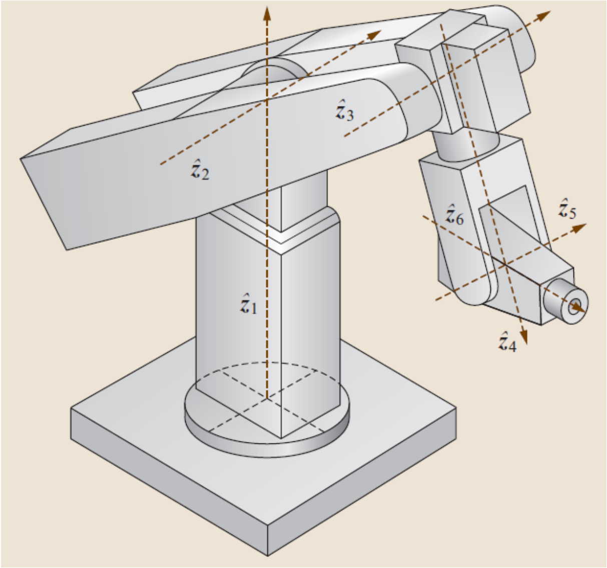


Figure 2.1: Example of a 6 DOF manipulator. [Siliciano and Khatib, 2008, chap. 1, page 24]

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 and Khatib, 2008]. 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

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 and Khatib, 2008, chap. 1]. In simple terms, the workspace is the space that can be reached by the robot.

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 and and 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.

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 threedimensional 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 and Khatib, 2008]. 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.

2.3.3 Motion Planning Algorithms

write more, add more references, aprox 30 at the end

One of the fundamental problems of robotics is to plan motions for complex bodies from an initial pose to a goal. As explained by Siliciano and Khatib [2008], 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 collision, 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 2.4.3), will be used as robot motion controller.

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

2.3.3.1 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.

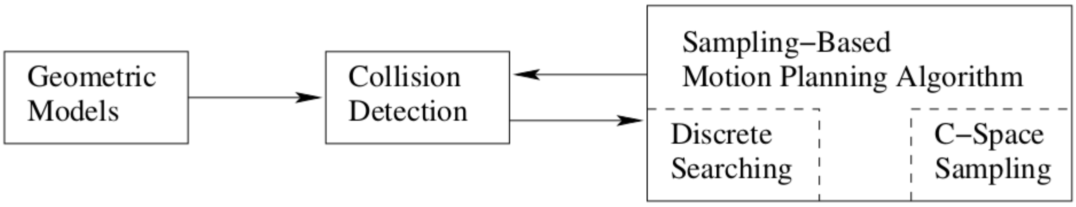


Figure 2.2: Sampling-based planning philosophy [LaValle, 2006, chap. 5, page 185].

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 and Khatib [2008], to solve such problems with algorithms that uses the C_{obs} directly.

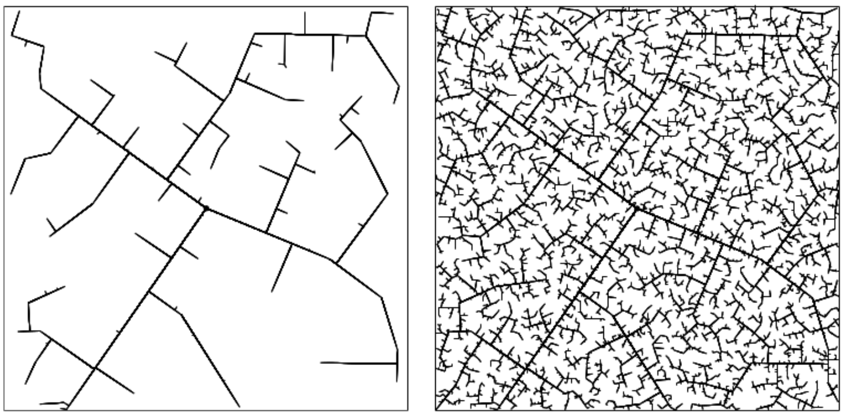
The disadvantage of these algorithms is that they don't assure to find a solution in a finite amount of time. Combinatorial motion planning algorithms are able to return a solution, if it exists, in a finite amount of time.

A **rapidly exploring random tree (RRT)** is a sampling-based planning algorithm that searches

Algorithm 1 Basic RRT

```
$\boldsymbol{G}.init(q_1)$;
for $i = 1$ to $K$ do
    $\boldsymbol{G}.add\_vertex(\alpha(i))$;
    $q_n \longleftarrow$ NEAREST$(S(\boldsymbol{G}, \alpha(i))$;
    $\boldsymbol{G}.add\_edge(q_n, \alpha(i))$;
end for
```

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.



45 iterations

2345 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 C_{free} between the initial and the desired configuration. There are two different approaches for finding this path (figure 2.4):

- **Single-tree search:** The planner grows a tree from q_1 as shown in figure 2.3 and checks in every step if it is possible to reach q_G with the S created by the tree.
- **Balanced, bidirectional search:** Instead of creating only one RRT, it creates an aditional tree that starts from q_G, this is quite effective when obstacles are partially surrounding q_G or q_1. The graph G is formed by two trees, T_a and T_b, growing from q_1 and q_G respectively. Here, S is formed by T_a and T_b. After some iterations, the trees are swapped, so T_b is now growing around 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.

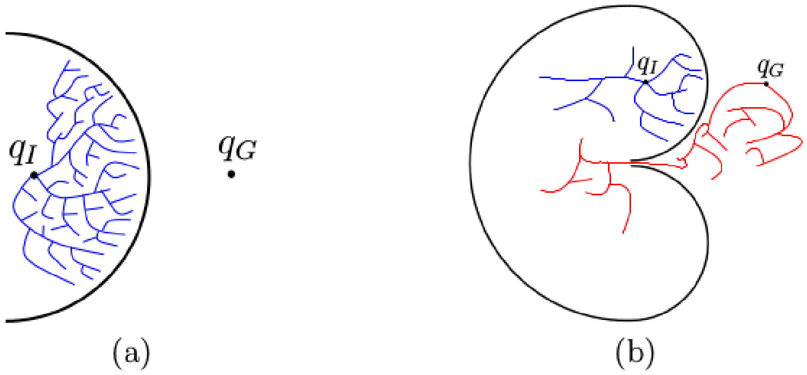


Figure 2.4: a) Single tree RRT, b) Bidirectional RRT [LaValle, 2006, chap.5, page 219].

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

2.3.3.2 Optimization-based motion planning

A different approach to solve the trajectory generation software, 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.

TODO

finish this section Sample-based VS Optimization-based

2.4 Software

In the last years, the number of applications where humans and robots work in proximity to each other has increased. 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, so this project was done using ROS as base. For trajectory planning, I used Giskard (section 2.4.3), with RVIZ as a side tool for visualization.

2.4.1 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 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

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

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.

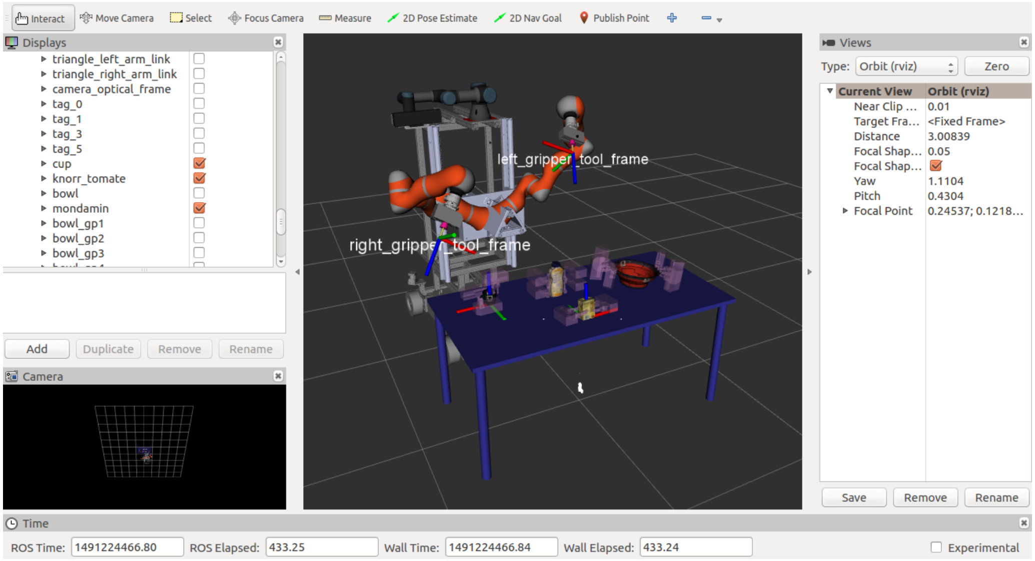
Since most applications require several nodes to control all the subsystems, it is useful to have a code that can launch multiple nodes locally and remotely. A *launch file* is a XML configuration file with a *.launch extension, it can specify a set of parameters and nodes to launch. These files can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

2.4.2 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 2.5), visualize simulated trajectories, and display information from the sensors, such as 3D

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

point-clouds from the Kinect or the image obtained by a camera.



- triangle_left_arm_link □
- triangle_right_arm_link □
- camera_optical_frame □
- tag_0 □
- tag_1 □
- tag_3 □
- tag_5 □
- cup ☑
- knorr_tomate ☑
- bowl □
- mondamin ☑
- bowl_gp1 □
- bowl_gp2 □
- bowl_gp3 □

Type: Orbit (rviz)

Current View	Orbit (rviz)
Near Clip ...	0.01
Target Fra...	<Fixed Frame>
Distance	3.00839
Focal Shap...	0.05
Focal Shap...	☑
Yaw	1.1104
Pitch	0.4304
Focal Point	0.24537; 0.1218...

ROS Time: 1491224466.80 ROS Elapsed: 433.25 Wall Time: 1491224466.84 Wall Elapsed: 433.24 □ Experimental

Figure 2.5: 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: robot's current state, location of objects in the environment, and data from sensors.

Figure 2.5 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 also be seen.

2.4.3 Giskard

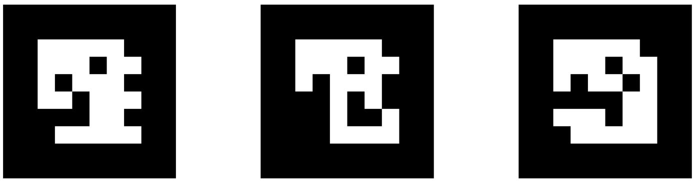
Giskard[5] is a constraint- and optimization-based framework for robot motion control developed by the Institute for Artificial Intelligence[6]. Giskard was used as base to develop the motion controller used in this work.

This controller allows using action models that inform the robot how to execute a specific motion, such as pouring or grasping.

[5] https://github.com/SemRoCo/giskard_core
[6] http://ai.uni-bremen.de/

2.4.4 Chilitags

Chilitags[7] 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.



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

Figure 2.6: Example of Chilitags fiducial markers, each tag has a different ID number.

Figure 2.6 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).

2.4.5 MoveIt!

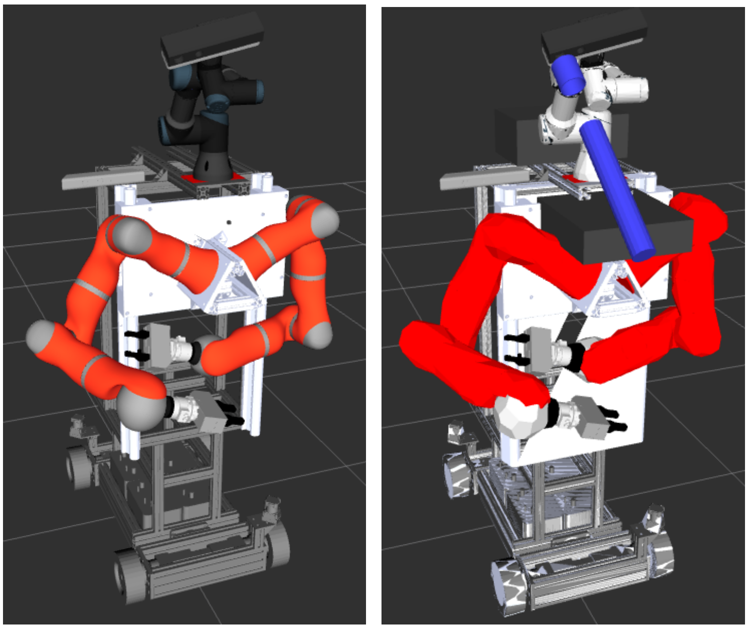
update figures!! collision model changed

[7] http://chili.epfl.ch/software

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 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 2.5). 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 2.7, one for visualization and one for collision.



(a) Visualization Model (b) Collision Model

Figure 2.7: Models of Boxy used by MoveIt!

The *visualization model* is used by RViz (section 2.4.2) to visualize the robot in the computer.

This model is normally as close as possible to the real robot, that means that the geometry tends to 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.

2.5 Robot Description

In oder to calculate the position and movements of the 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.

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 better grasping pose according to the initial conditions.

A general description of the system's architecture is shown in figure 3.1. The system uses ROS (section 2.4.1) 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.

update diagram

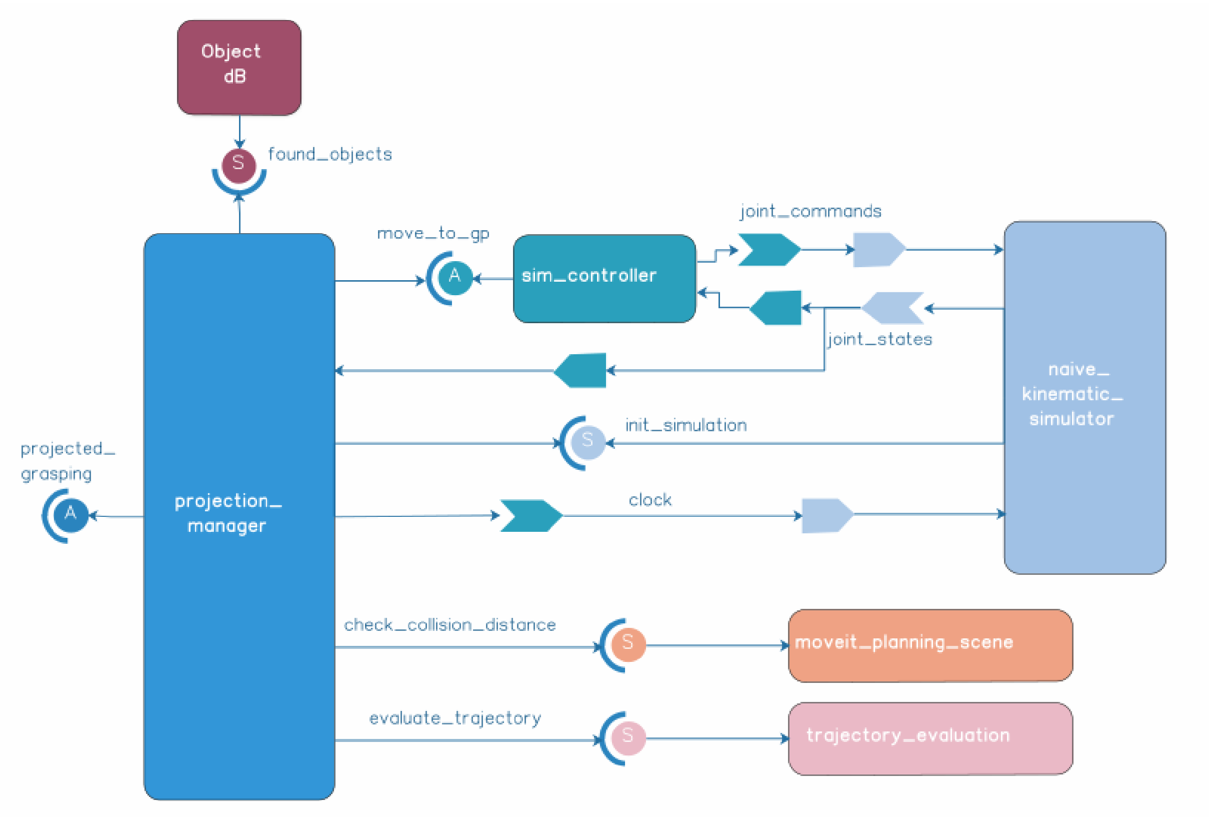


Figure 3.1: System architecture: The perception system (*Object_dB*) sends location of found objects to the system manager (*projection_manager*), which requests a trajectory to the controller (*sim_controller*).

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.

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 grasp it. Then, it sends the trajectories to the *trajectory_evaluation* node, which evaluates between several proposed trajectories, selects one and sends it back as result.

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.

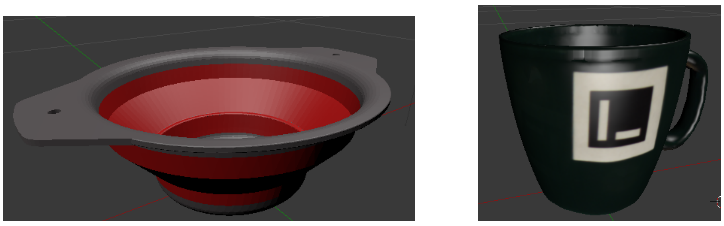


Figure 3.2: DAE models of a bowl and a cup. In the cup, one of the markers used for perception can be seen.

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, it's 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.

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

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:

```
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
  scale: 1.0                # if the model is in cm = 1, m = 0.001
  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.

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.

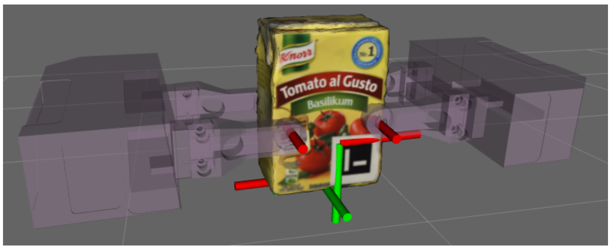


Figure 3.3: 3D model of a tomato sauce package with two grasping poses. The object and GP have reference frames.

Figure 3.3 shows an object with two predefined GP. 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 2.4.3). Giskard uses *qpOASES*[4] to calculate a trajectory for each joint. As stated by Ferreau et al. [2014], qpOASES is an open-source optimization software that can solve quadratic programming (QP) problems of the form:

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

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 following constraints:

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

$$l\vec{b}A(\omega_0) \;\leq\; \mathbf{A}\vec{x} \;\leq\; u\vec{b}A(\omega_0)$$

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

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 $\vec{lbA}$, $\vec{ubA} \in \Re^{nV}$. 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 potion 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:

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

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

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

$$\begin{pmatrix} -\dot{q}_{0,max} \\ -\dot{q}_{1,max} \\ -\varepsilon_{max} \end{pmatrix} \leq \begin{bmatrix} \omega_1 & 0 & 0 \\ 0 & \omega_2 & 0 \\ 0 & 0 & \omega_3 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon \end{pmatrix} \leq \begin{pmatrix} \dot{q}_{0,max} \\ \dot{q}_{1,max} \\ \varepsilon_{max} \end{pmatrix} \tag{3.16}$$

and

$$\begin{pmatrix} error \\ -q_{0,max} - q_0 \\ -q_{1,max} - q_1 \\ a_{0,min\_lim} \\ a_{1,min\_lim} \end{pmatrix} \leq \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \varepsilon \end{pmatrix} \leq \begin{pmatrix} error \\ q_{0,max} - q_0 \\ q_{1,max} - q_1 \\ a_{0,max\_lim} \\ a_{1,max\_lim} \end{pmatrix} \tag{3.17}$$

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. The initial velocity of the joints is assumed to be zero, but other initial conditions could also be set.

3.3.1 Simple 2 DOF controller

A proportional gain 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:

$$error \; = \; p \, * \, \left(q_{eef,des} - q_{eef}\right) \tag{3.18}$$

Figures 3.4 and 3.5 show the position and velocity of a 2 DOF manipulator, whose trajectory is calculated using equations 3.16 and 3.17 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.

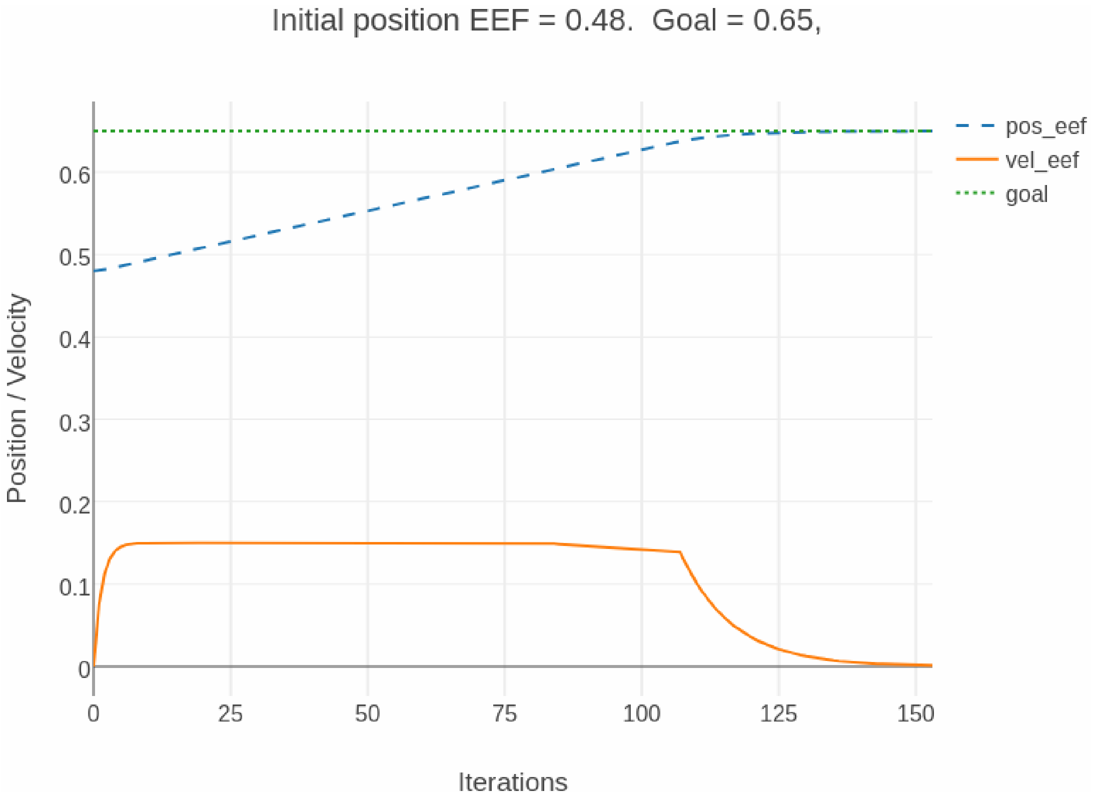
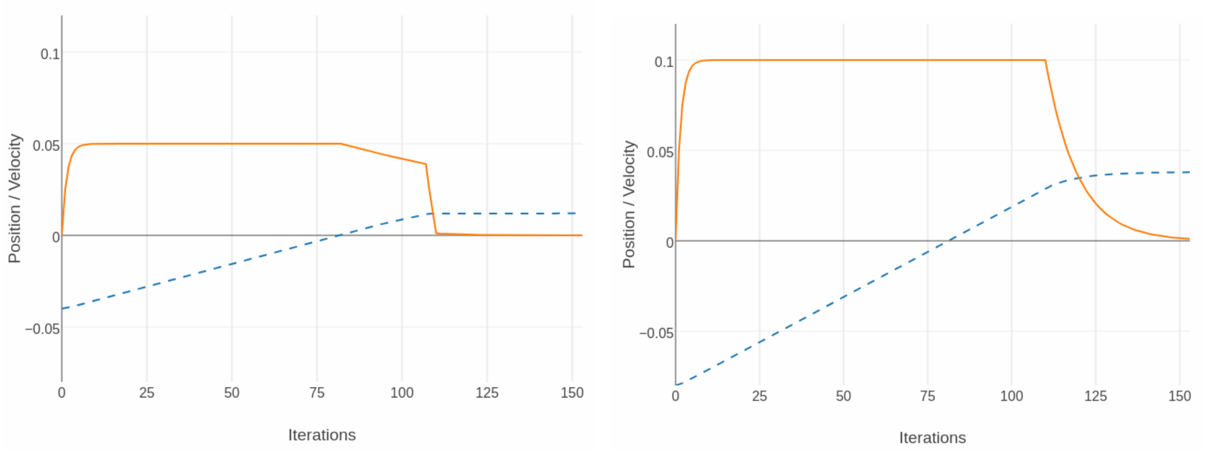


Figure 3.4: EEF trajectory. Initial acceleration is limited by constraints. Velocity decreases as EEF aproaches the goal position.



(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 equations 3.14 and 3.15. The maximum acceleration is calculated on every iteration using

$$a_{n,max\_lim} = \frac{\dot{q}_{n,max} - a_{n,max}}{\dot{q}_{n,max}} + a_{n,max} \tag{3.19}$$

$$a_{n,min\_lim} = \frac{\dot{q}_{n,max} - a_{n,max}}{\dot{q}_{n,max}} - a_{n,max} \tag{3.20}$$

where n is the joint number and $a_{n,max}$ a constant acceleration limit.

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 3.17 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 $joint\_error = q_{0,des} - q_0$.

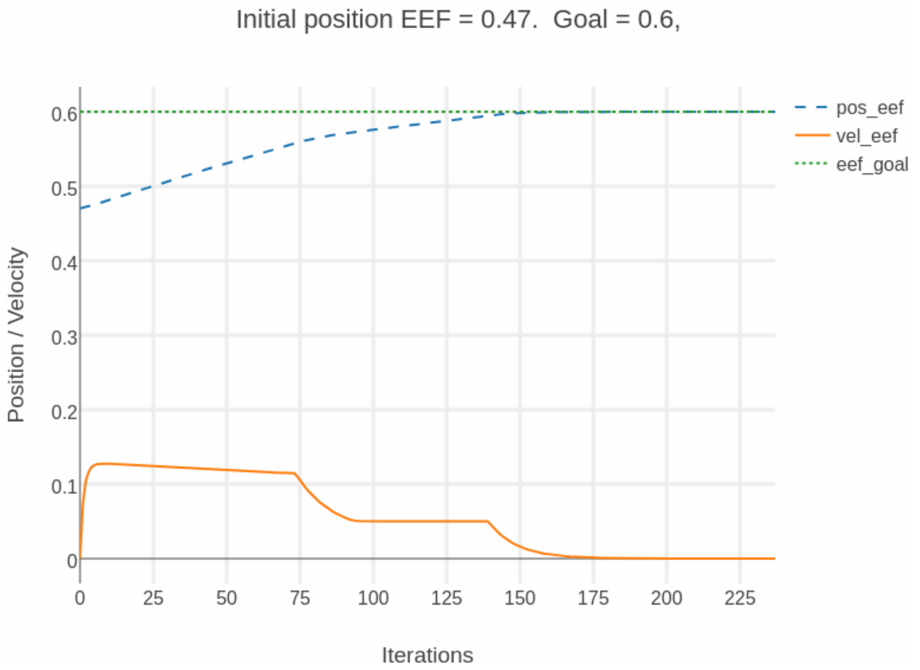
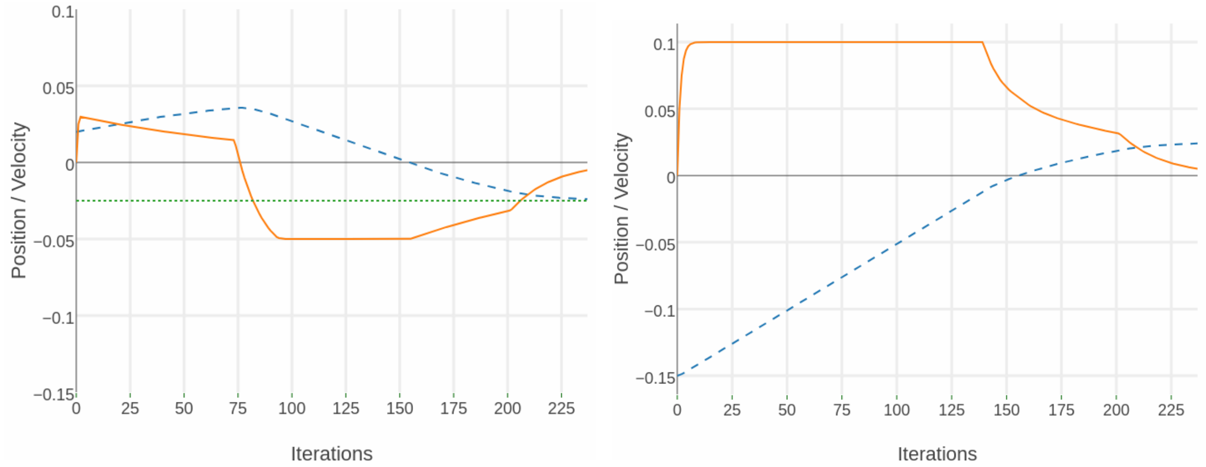


Figure 3.6: Trajectory (dashed), velocity (continuous) and goal (dotted) of the EEF.

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.

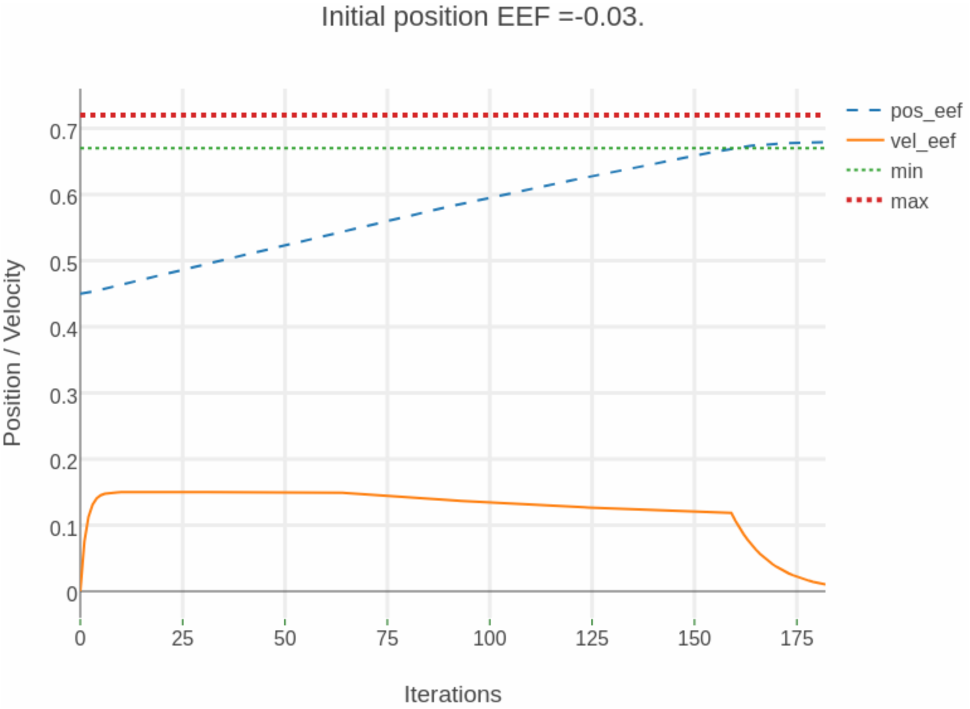


Figure 3.8: Trajectory (dashed) and velocity of the EEF, the goal range is between the dotted lines.

To achieve this behavior, matrix **A** and it's limits, $l\vec{b}A$ and $u\vec{b}A$, are modified, rewriting equation 3.17 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. 3.16) 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.

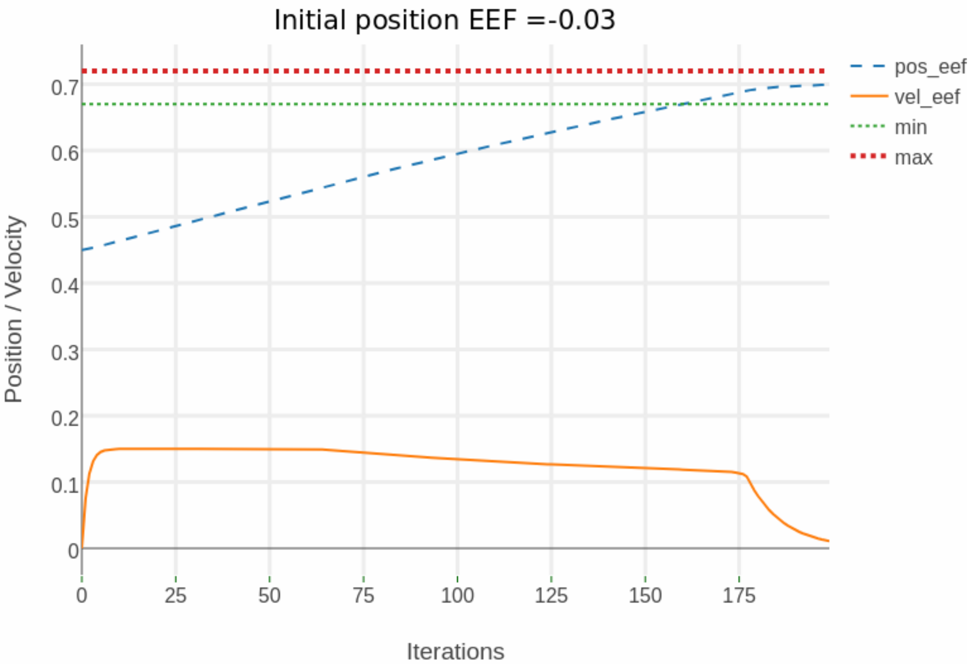
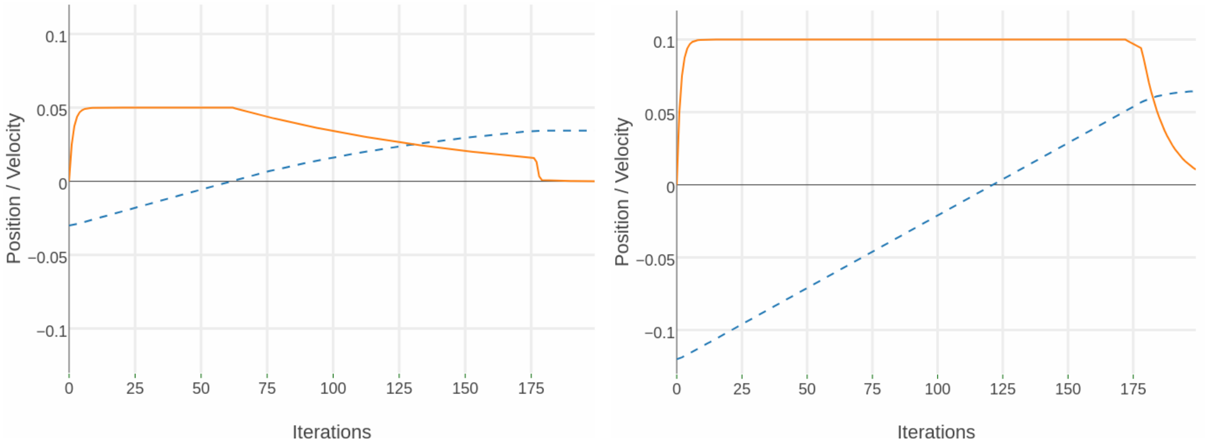


Figure 3.9: Trajectory (dashed) and velocity of the EEF, here an attractor is defined inside the range.



(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

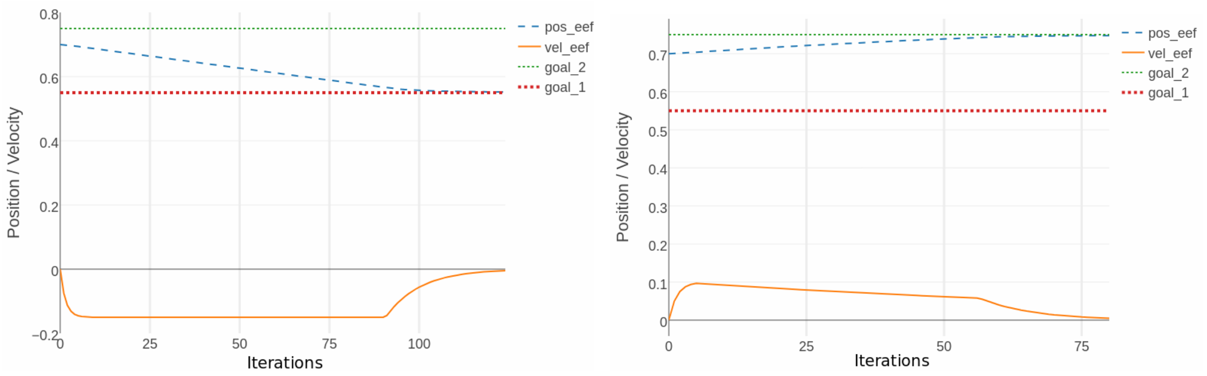
It is also possible to specify multiple goals for the EEF and select one of them using the weights assigned to each goal. This controller shows a two DOF linear manipulator where two goals are specified. In this case, the matrix **A** and it's limits (given by equation 3.17) are given by:

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

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. 3.16) 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).

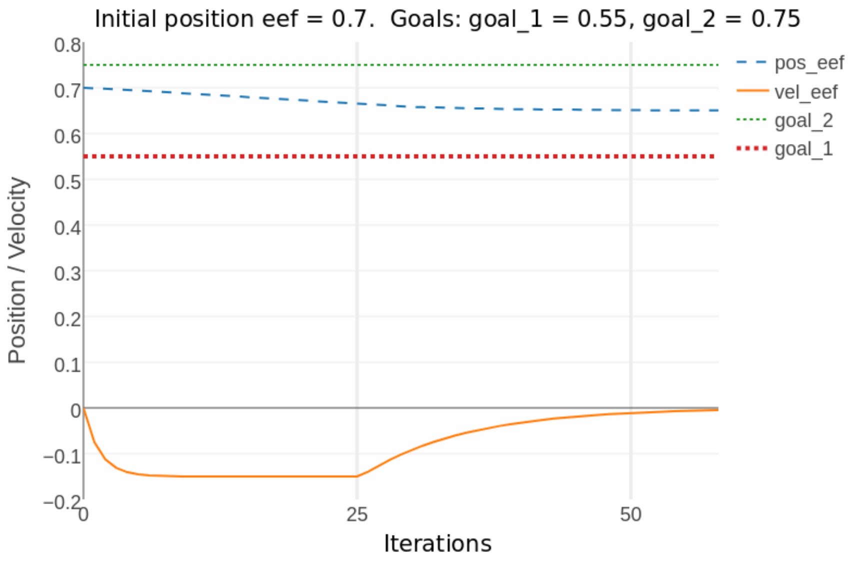


Figure 3.12: Trajectory (dashed) and velocity of the EEF. Same weight in both goals. Each joint has a different velocity and acceleration constraint, as well as different joint limits.

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 goals' weight in every iteration, with the error to the goals q_{des1} and q_{des2} as:

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

and

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

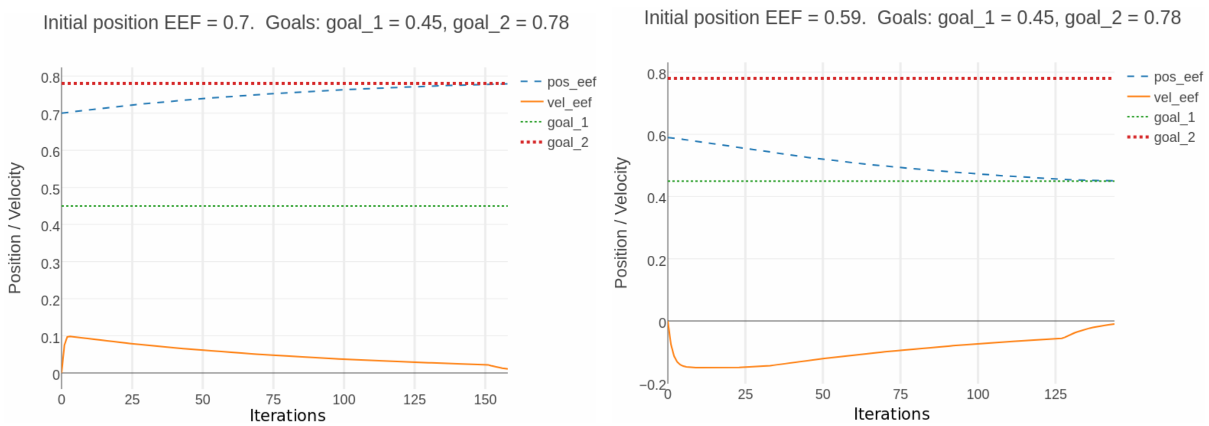
where p is the proportional gain and q_{eef} the current position of the EEF. The weights are calculated with:

$$\omega_3 = \left(\frac{error_2}{error_1 + error_2} \right)^2 \tag{3.21}$$

$$\omega_4 = \left(\frac{error_1}{error_1 + error_2} \right)^2 \tag{3.22}$$

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

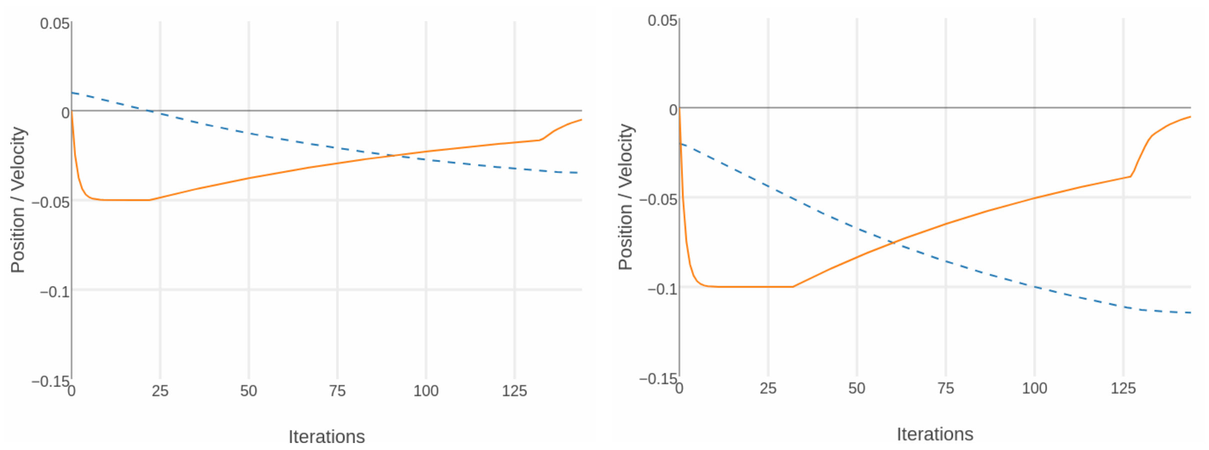
In this example, the initial position of the EEF is set between both goals. The first goal is set at 0.45 and the second one at 0.78. As shown in figure 3.13, the EEF reaches the closes goal. The only parameter changed between figure 3.13 (a) and (b) is the EEF's initial position.



(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 3.22 as:

$$\begin{aligned}\omega_3 &= \left(\frac{1-|error_1-error_{min}|}{error_{max}+error_{min}}\right)^3\\ \omega_4 &= \left(\frac{1-|error_2-error_{min}|}{error_{max}+error_{min}}\right)^3\\ \omega_5 &= \left(\frac{1-|error_3-error_{min}|}{error_{max}+error_{min}}\right)^3\end{aligned} \tag{3.23}$$

where:

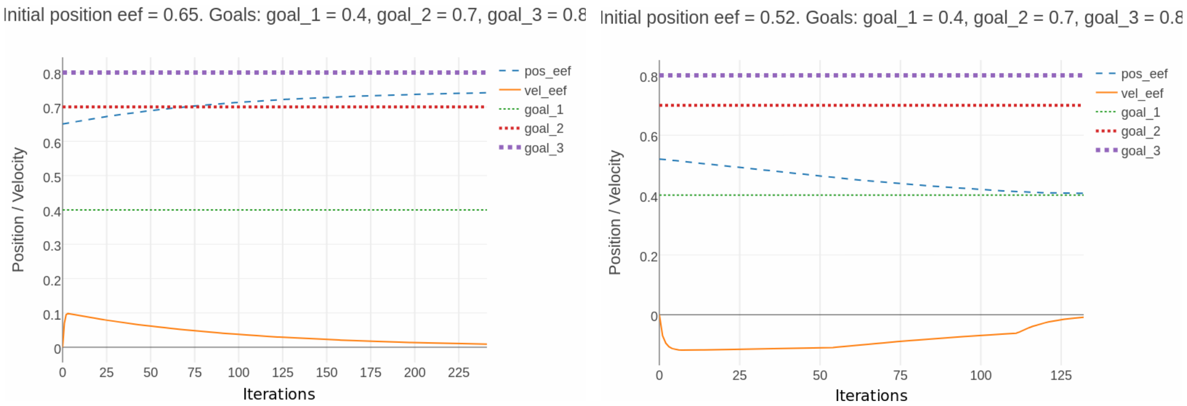
$$error_{min} = min(|error_1|,|error_2|,|error_3|)$$

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

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

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

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 showed in the previous subsections was developed for the Boxy robot. Boxy's geometry and kinematics are taken from the URDF file (section 2.5).

The definition of the matrix **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:

$$\vec{s} = [\,\dot{q}_0\ \ \dot{q}_1\ ...\dot{q}_C\ \ \varepsilon_0 ...\ \varepsilon_6]^T \tag{3.24}$$

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:

$$\mathbf{A} = \begin{bmatrix} \mathbf{J} & \mathbf{I}_{p,p} \\ \mathbf{I}_{m,m} & \mathbf{0}_{m,p} \\ \mathbf{I}_{m,m} & \mathbf{0}_{m,p} \end{bmatrix} \tag{3.25}$$

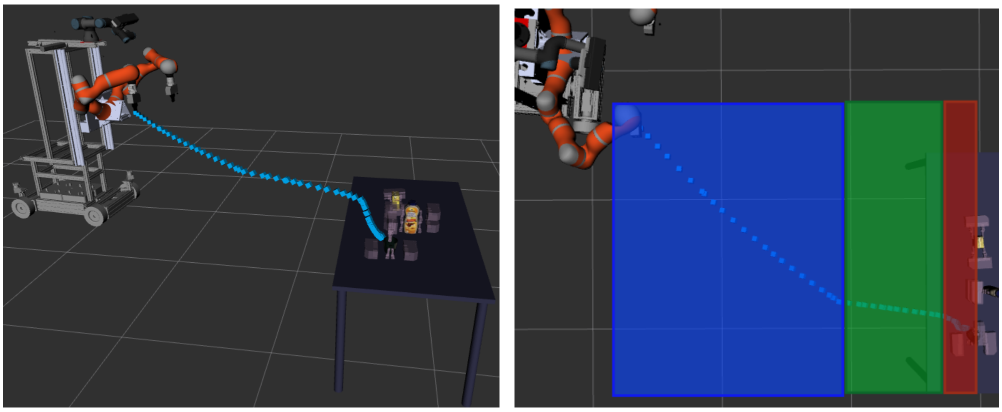
where $\mathbf{J} \in \Re^{p \times m}$ is the Jacobian, $\mathbf{I}$ is the identity matrix, $\mathbf{0}$ is a zero matrix, p is the number of DOF and m the number of joints to control. The constraint vectors are created using the error, joint limits and acceleration constraints:

$$l\vec{b}A = [\ pos\_\vec{e}rror,\ orien\vec{t}\_error,\ -\vec{q}_{max} - \vec{q},\ \vec{a}_{min\_lim}]^T$$

$$u\vec{b}A = [\ pos\_\vec{e}rror,\ orien\vec{t}\_error,\ \vec{q}_{max} - \vec{q},\ \vec{a}_{max\_lim}]^T$$

where the acceleration limit is calculated using equations 3.20 and 3.19, $\vec{q}_{max}$ is the vector of joint limits and $\vec{q}$ the current position of the joints. The $pos\_\vec{e}rror$ is the position error and $orien\vec{t}\_error$ the orientation error.

The weight vector $\vec{\omega}$ is calculated on every iteration. 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 wight 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 (blue) 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 left area in figure 3.16 (b) shows the part of the trajectory where only the base was moved, the middle area is where arms, torso and base were active and the right area is where only the arm and torso were used.

The first goal set to the controller is a pre-grasping pose, located three 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.

Write some more about the controller

3.4 Collision Checking

write!!!

3.5 Hardware and Software requirements

improve this section

This project was developed in a computer running on Ubuntu 14.04, no special hardware is required to execute the generated code.

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

- **ROS:** With a configured workspace. The ROS distro used was *indigo*
- **MoveIt!:** Requires previous installation of ROS
- **UR3 driver:** A recommended one is `ur_modern_driver`
- **Boxy repositories:** The `iai_boxy` and `iai_robots` from the IAI github repository[6]
- **UR3 MoveIt configuration:** The code generated as a result of this project[7]

After installing ROS and MoveIt!, the user has to download the specified repositories from Github.

[6] `https://github.com/code-iai`

[7] `https://github.com/mgvargas/Moveit_config`

CHAPTER 4

Evaluation

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. In order to measure the quality of the trajectories obtained by the controller and decide if the gasping action was successful, these trajectories must be evaluated.

4.1 Experimental Setup

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, 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 send it to the robot.

The metrics considered for each trajectory are:

- Length of the generated path
- Smoothness: No abrupt changes in velocity and acceleration

- Convergence error: How far was the EEF 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
- Planning time: How long did the controller required to calculate the trajectory

All trajectories that generate a collision or that do not reach a certain threshold around the grasping pose are discarded. The remaining trajectories are scored based on these metrics and the best one is sent to the robot.

4.1.1 Scenario 1: Simulation with objects our of reach

The scenario simulated three objects detected on top of a table: a cup, a tomato sauce package and a bottle of pancake mix. A given initial position for the robot was loaded in this environment. The robot was positioned far away from the table, so that the objects were out of reach for it. This was done in order to test the base's movement.

4.1.2 Scenario 2: Simulation with objects within reach

This scenario contained the same objects as the first one, but here the robot was placed closer to the table, so that the arms and torso started moving from the beginning of the trajectory.

4.1.3 Scenario 3: Simulation with multiple goals

The controller allows to specify of the robot should reach only one grasping pose, or any point in a region between two grasping poses. This feature can be useful when grasping objects like a plate or a bowl, that can be grasped from any point on the border or handle. so reaching a specific point is not required.

4.1.4 Scenario 4: Testing with the robot

The last experiment was executed on the real robot. The robot was placed in from of a table with one of the objects from the database. In this case, the trajectory evaluation included the execution time (the time the robot required to execute the given trajectory).

4.2 Experimental Results

The first experiments were made in simulation and visualized using RVIZ (section 2.4.2). In all experiments, grasping is considered successful if the robot's EEF reached one of the defined grasping poses without collisions.

reaching or lifting object for a couple of seconds??

4.2.1 Scenario 1: Simulation with objects our of reach

What happened? rosbag, plots, table of evaluation

4.2.2 Scenario 2: Simulation with objects within reach

4.2.3 Scenario 3: Simulation with multiple goals

4.2.4 Scenario 4: Testing with the robot

CHAPTER 5

Discussion

APPENDIX A

Appendix

A.1 Github Repositories

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`

- Robot's kinematic simulator:

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

.

.

.

A.2 Code Execution

explain how to run the code. Required?

A.2.1 Perception system

A.2.2 Robot

A.2.2.1 Target: Single Goal (or One-hand grasping)

A.2.2.2 Target: Multiple goals (or Two-hand grasping)

Bibliography

Sachin Chitta, Ioan Sucan, and Steve Cousins. Ros topics, moveit! *IEEE ROBOTICS AND AUTOMATION MAGAZINE*, pages 18–19, 2012.

John J. Craig. *Introduction to Robotics; Mechanics and Control*. Pearson Education International, third edition, 2005.

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

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

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

Steven M. LaValle. *Planning Algorithms*. Cambridge, 2006.

R. A. Russell. *Robot Tactile Sensing*. Prentice Hall, 1990.

Bruno Siliciano and Oussama Khatib. *Springer Handbook of Robotics*. Springer, 2008.

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