Proximity-informed robot control through whole-body artificial skin

by

A. Aranburu Fernndez

B.A., UPV/EHU Euskal Herriko Unibertsitatea, 2017

M.S., UPC Universitat Politecnica de Catalunya, 2020

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2020

This thesis entitled:
Proximity-informed robot control through whole-body artificial skin
written by A. Aranburu Fernndez
has been approved for the Department of Computer Science

_____

Prof. Alessandro Roncone

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Aranburu Fernndez, A. (M.S., Industrial Engineering)

Proximity-informed robot control through whole-body artificial skin

Thesis directed by Prof. Alessandro Roncone

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Contents

Chapter

Chapter 1

Introduction

1.1 Human-Robot Interaction

We are on the threshold of a new era in robotic technology that will bring breaking improvements into our lives. Simple household robots, such as the vacuuming robot Roomba® [1] or the kitchen robot thermomix® [2], are starting to do some of human's the most time consuming tasks. But this is just the beginning and robots will soon be more extended in our daily lives. This robotic revolution we are talking about has the potential to surpass the ubiquity and usefulness of the computer revolution, if only because robots can change not only our virtual world, but the physical world also.

Human-robot interaction (HRI) is a field of study dedicated to understanding, designing, and evaluating robotic systems for use by or with humans. It is one of the main fields of research working towards the revolution presented in the first paragraph and it is focused on expanding robot's abilities to interact with the physical world, integrating robots in human-populated environments. There are a number of ways in which robotics can be impactful to society in the short- mid- term. Some examples:

- Hospital robots will check on patients and report their status to nurses, saving time and improving patient outcomes. One current example is the Moxi® hospital assistant robot [3]. It can already accomplish tasks such us gathering supplies and bringing them to patient rooms and delivering lab samples.

- Childcare robots will help parents with chores such as assisting at diaper changing or feeding, so that families can spend high-quality time together. iPAL® [4] is a sneak peak of what is to come.

- Manufacturing robots will collaborate with people to assemble complex objects on reconfigurable assembly lines, increasing the efficiency and flexibility of factory floors.

Nowadays robots in the industry live in restricted access areas and interaction with humans is avoided. The transition of robots from these restricted access areas to human-populated environments comes with a number of problems that need to be overcome. One of the key issues is that human safety is compromised. Traditionally, safety in industrial robots has been achieved isolating the robotic platforms in a cell with safety interlocks to prevent direct interaction. HRI contributions will only be feasible once the coexistence of robots and humans does not come with serious injury risk.

1.2 Human senses and sensors in robotics

In order to achieve the safety requirements of HRI, robots are expected to adapt to the changes in their environment the way humans do. This is why it is important to study how humans interact with their environment.

Interaction between a human being and its environment is accomplished through its senses. One important aspect research in robotics has focused on has been to look for ways to emulate and exploit these human senses. For example, visual sensing has been widely explored in this field. One of the reasons is the development of new low-cost depth sensors such as the Microsoft Kinect ™ has allowed to meet the many requirements of vision application at an affordable price. These depth sensors are capable of acquiring 2D images as well as the distances of the objects represented in each pixel to the camera. The usual way this data is used is to reassemble representations of obstacles in a robot-oriented space.

However, other human senses are of great importance for the acquisition of data relevant

to the interaction with their surroundings. Touch is one of them. Tactile sensing is the most fundamental sense when contacting the external world and it is the biggest and oldest of the sensorial organs. An experiment that consisted of exploring objects after anesthetizing the hands of a control group demonstrated that the simple task of maintaining a stable grasp of objects becomes surprisingly difficult when sense of touch is suppressed. Their hand movements are inaccurate and unstable [5]. In another experiment carried on by astronauts at the International Space Station, sense of touch was proven to be an important indicator of direction and spatial disorientation [6]. Sense of touch is crucial when experiencing object properties, such as size, shape, texture and temperature. It informs about slip. It is essential to develop awareness of the body and in consequence, to differentiate ones body from the rest of ones surroundings. Its absence seriously hinders the interaction of an individual with its environment [7].

Nonetheless, tactile sensing in robotics has not been able to achieve the penetration this evidence suggests it should have had. In traditional industrial robots, this importance has been ignored. Engineers have been able to avoid the issue of developing a artificial sensor that emulates touch by using prior knowledge about the object to be manipulated and the environment. The limitations of this approach are obvious: robots are only capable of working in structured and controlled environments. https://www.sciencedirect.com/science/article/pii/S0921889015001621

The result is that research and technology in artificial tactile sensors is not as well developed as other perception modalities. Tactile sensors in robotic applications are represented by:

- **Pressure sensing arrays.** Pressure sensor matrix that provides information about the location and amount of pressure exerted on a surface.
- **Force-torque sensors.** They give feedback about the forces and torques that are applied on a given point in the 3 geometric axes.
- **Light level change detection** https://www.wired.com/story/this-clever-robotic-finger-feels-with-light/

1.3 Whole-body artificial skins

Current research is focused on developing tactile skins that cover robot end effectors and hands with a number of tactile sensors, in parallel with devising new algorithms that make use of these new sensors for dexterous object manipulation ([8], [9]).

Even though we know about the importance of human skin covering all the body, whole-body artificial skins have not gained the same attention as end effector coverings. In fact, these types of skins can be of great importance in the transition of robots to operating alongside humans and the new safety issues that arise with it.

Traditional robots have been designed for precision and performance at expenses of safety and the ability to adapt to the surroundings. The two paradigms that try to find a solution to this issue, different but not mutually exclusive, are: **passive** and **active compliance**. Passive compliance aims to achieve safety through physical adaptation to the environment. There are two different strategies to accomplish that. On the one hand, there is the use offlexible parts and mechanisms. On the other hand, limitation of the performance capabilities of the robot, such us speed, can also be enough to achieve safety. Active compliance, in contrast, relies on algorithms and control laws for reactive interaction with the environment.

Both approaches have some important limitations, though. Passive compliance achieves safety at the cost of reducing accuracy, in the case of using flexible parts, and agility when the capabilities of the robot are reduced. As far as active compliance is concerned, sensors that are used are far from being perfect for the control. Some sensors provide either a high amount of low-quality information regarding the task, as in the case of the aforementioned depth cameras or a limited amount of relevant data, as force-torque sensors located at the robots joints. In addition, both solutions are currently lacking in flexibility. New problems require completely new designs. These are important enough reasons that explain why none of the solutions have become the established standard in robotics.

Whole-body artificial skins have the potential to solve all the issues present in active and

passive compliance. It does so providing high-density relevant information, information that is much more relevant to the task of collision avoidance. At the same time, it can also be designed to be mechanically compliant to the environment.

In addition, the skin can be equipped with heterogeneous sensing so that a greater amount of high-quality information is collected. Tactile data is only useful when contact has been established between the robot and the environment. However, collisions can be actively avoided before establishing undesired contact by choosing suitable pre-impact strategies. That is why proximity sensing opens up new possibilities for the task of collision avoidance. The addition of proximity sensors allows a compact, external motion capture free system, with highly informative prior-to-contact data. This combined with lighter data, makes proximity sensing the perfect sensing system for the solution of the collision avoidance problem.

1.4 Control and collision avoidance

From the control perspective, using proximity data for the safe movement of a robot might prevent performing the task commanded to a robot, that is, the original trajectory when there were no impediments that prevented the desired movement commanded to perform a certain task. When any undesired object obstructed the desired trajectory of a robot, it is not difficult to see that the performance of that main task could be hindered. Nonetheless, this does not have to be the case for a certain type of robots called redundant robots.

A robot is task redundant if the number of degrees of freedom (DoF) n it has is greater than the degrees of freedom needed for the task m. For example, if the specifications of a given task only require positioning the end-effector of a robot at a certain point, i.e. $m = 3$ a robot with $n = 3$ DoF would be able to accomplish the task and one with $n = 4$ DoF would be task redundant. A redundant robot has the advantage of being able to accomplish the task in infinitely many possible ways.

The ability to perform a certain movement of the end effector in infinitely many different ways means that safe motion control does not imply stopping the main task and in fact, the main

task should only halt when there is no way to make use of redundancy to avoid a collision while completing it.

1.5 Objectives and scope

Chapter 2

Related Work

Introduction talking about the two subsections of related work. Why we care.

2.1 Whole body articial skins

Introduce the kinds of skins. We are using something like mittendorfer that is kind of an individual skin unit.

2.2 Avoidance control

Alessandro's paper

Look cites

Chapter 3

Technical Approach

In this chapter, we will explain the technical means by which the objective of this thesis has been approached. Firstly, in section 3.1 we start talking about the basic elements that have been used during the development this work, such us the robot platform employed or software libraries. In order to make the robot avoid obstacles with proximity data, obstacles need to be located with respect to the robots kinematic chain. That's why, in section 3.2 we present the method to sense the environment through proximity sensing. Lastly, in section 3.3 where the methods for collision avoidance are explained.

3.1 Required resources

In this section we present the various types of tools that have employed throught the process. We start presenting the robotic platform employed for carrying on the experiments. Then we define what robotic middleware is, why it is useful and we present the specific robotic middleware that have been employed: ROS. We finish this section presenting the several software libraries that have been useful in the implementation of the concepts and ideas that will be presented in the sections that follow.

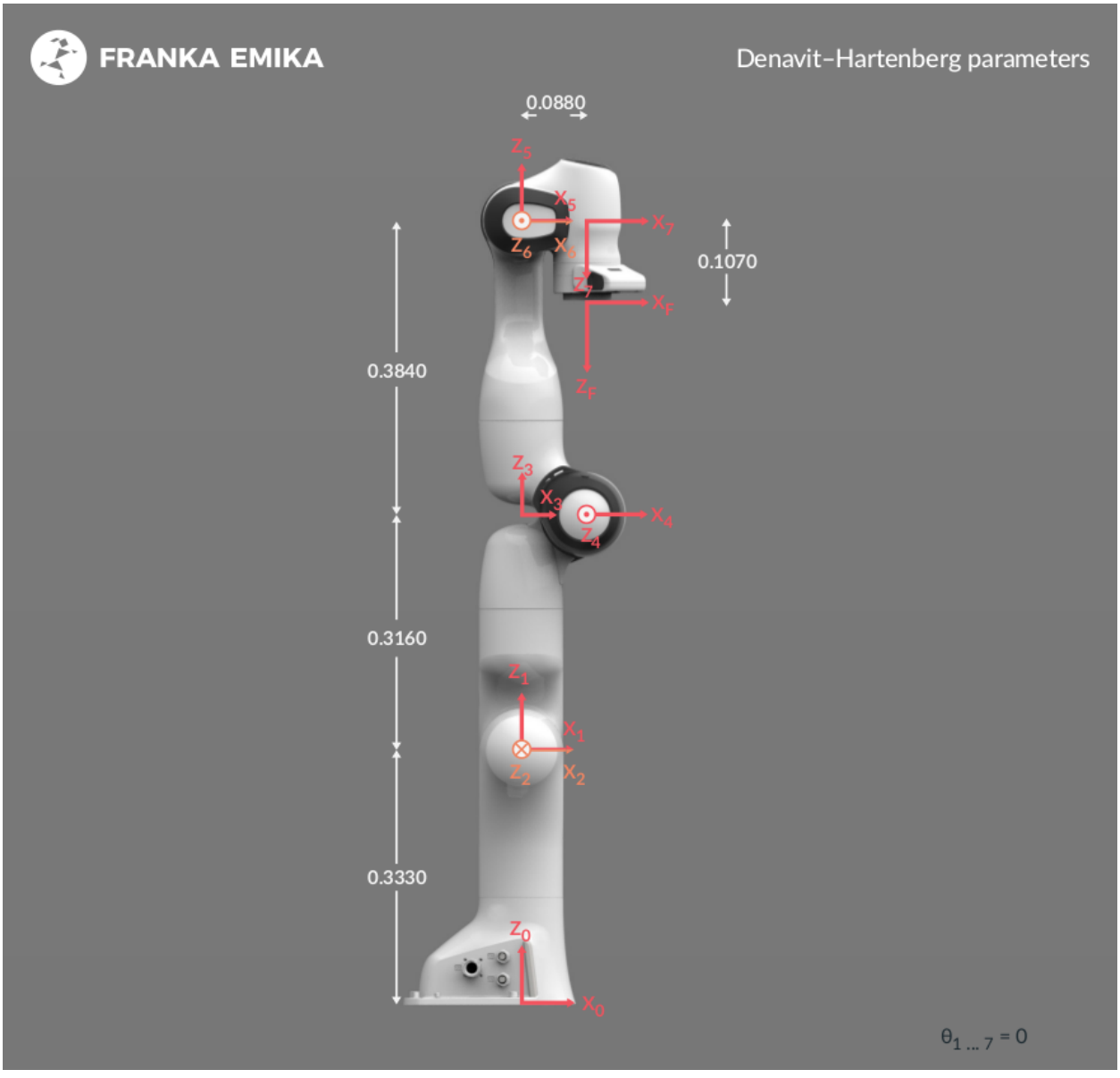
3.1.1 Robot platform

The robot utilized for the development of the ideas presented in this work has been the Panda, manufactured by the company Franka Emika based in Munich, Germany. It is well known

for its performance and usability as well as for its affordability, starting at 10,000$.

The Franka Emika Panda (FEP) is equipped with 7 revolute joints, i.e. it has 7 degrees of freedom (DOF). Each of the revolute joints mounts a torque sensor. The total weight of FEP is around 18 kg. It can handle payloads of up to 3 kg.

Figure 3.1: FEP: Joints and dimensions [10].



It has a control interface that provides access to joint positions $\mathbf{q}$ and velocities $\dot{\mathbf{q}}$ and link side torques vector τ, with a refresh rate of 1 kHz. The control interface also provides numerical values of the inertia matrix $\bar{\mathbf{M}}(\mathbf{q})$, the gravity vector $\bar{\mathbf{g}}(\mathbf{q})$, the end effector Jacobian matrix $\bar{\mathbf{J}}(\mathbf{q})$ and the Coriolis term $\overline{\mathbf{c}}(\mathbf{q}, \dot{\mathbf{q}})$.

There are 3 control modes available. At the lowest level, there is the torque-mode $\tau_{\mathbf{d}}$. The joint velocity $\dot{\mathbf{q}}_{\mathbf{d}}$ and joint position $\mathbf{q}_{\mathbf{d}}$ modes provide a higher level control and are the ones that are going to be used throughout this work.

The Denavit Hartenberg (DH) parameters provided by the manufacturer [10] and depicted in figure 3.1 are shown in table 3.1.

Table 3.1: Denavit Hartenberg parameters [10]

Joint	**a** (m)	**d** (m)	$\boldsymbol{\alpha}$ (rad)	$\boldsymbol{\theta}$ (rad)
1	0	0.333	0	θ_1
2	0	0	$-\frac{\pi}{2}$	θ_2
3	0	0.316	$\frac{\pi}{2}$	θ_3
4	0.0825	0	$\frac{\pi}{2}$	θ_4
5	-0.0825	0.384	$-\frac{\pi}{2}$	θ_5
6	0	0	$\frac{\pi}{2}$	θ_6
7	0.088	0	$\frac{\pi}{2}$	θ_7
F	0	0.107	0	0

The manufacturer also provides the physical limits of the robot, both in the joint and the cartesian spaces (tables 3.2 and 3.3 respectively).

Table 3.2: Joint space limits [10]

	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6	Joint 7	
$\mathbf{q_{max}}$	+2.8973	+1.7628	+2.8973	-0.0698	+2.8973	+3.7525	+2.8973	rad
$\mathbf{q_{min}}$	-2.8973	-1.7628	-2.8973	-3.0718	-2.8973	-0.0175	-2.8973	rad
$\mathbf{\dot{q}_{max}}$	2.1750	2.1750	2.1750	2.1750	2.6100	2.6100	2.6100	$\frac{\text{rad}}{\text{s}}$
$\mathbf{\ddot{q}_{max}}$	15	7.5	10	12.5	15	20	20	$\frac{\text{rad}}{\text{s}^2}$

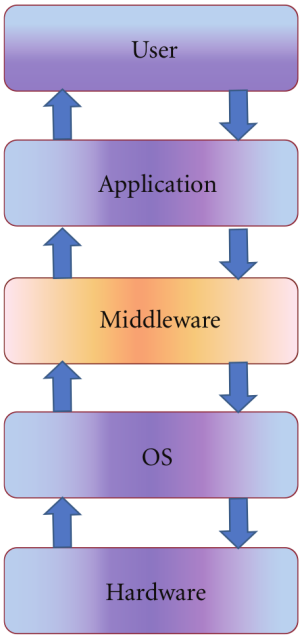
Table 3.3: Cartesian space limits [10]

	Translation	**Rotation**	**Elbow**
$\mathbf{\dot{p}_{max}}$	1.7000 $\frac{\text{m}}{\text{s}}$	2.5000 $\frac{\text{rad}}{\text{s}}$	2.1750 $\frac{rad}{\text{s}}$
$\mathbf{\ddot{p}_{max}}$	13.0000 $\frac{\text{m}}{\text{s}^2}$	25.0000 $\frac{\text{rad}}{\text{s}^2}$	10.0000 $\frac{rad}{\text{s}^2}$

3.1.2 ROS

The Robot Operating System (ROS) is an open source robot middleware, robot framework or robot development environment (RDE), one of many available in the market. Robot middleware [11] is an abstraction layer that resides between the operating system (OS) and software applications (figure 3.2). Its purpose is to provide a framework and take care of several important parts of applications development, so that the developer needs only to build the logic or algorithm as a component.

Figure 3.2: Robotic middleware [11]



The main goal of ROS in particular is to provide a framework for easy code reuse in research. This is why ROS is most extended in the academic context. It also provides APIs for Python and C++ and codes written in both languages can be used interchangeably and at the same time.

ROSs main concept is its runtime graph: a peer-to-peer network of loosely coupled components or nodes that use the ROS communication infrastructure. ROS implements two main communication types:

- **Topics.** A publisher node publishes a message to a topic. Another node subscribes to this topic, meaning that every time there is a new message published in the topic this subscriber node will receive it and therefore will we able to utilize and manipulate the data. Several nodes can publish to a given topic, as well as several nodes can also be subscribers of the same topic.

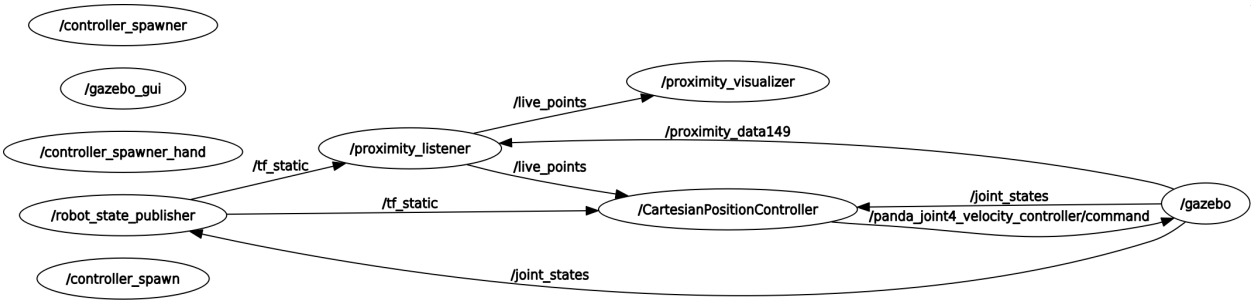
- **Services.** Sometimes the asynchronous model topics provide is not suitable. When synchronous behavior is desired services are the way to go. Services have two sides: client and

service. From the client side calling a service is equivalent to calling a function. However, this function is managed in a different node: the server side.

In figure 3.3 the runtime graph of this work is presented as an example. The real robot is replaced by a simulation. The /gazebo node is calculating the physics of the real robot, as well as sensor readings. /CartesianPositionController is subscribed to the topic /joint_states, as it needs this information for control purposes. Sensor data is processed in the /proximity_listener node and then is published to the topic /live_points. This topic has two subscribers in these examples. On the one hand, /proximity_visualizer uses the data published in /live_points to visualize the sensor data in rviz (the 3D visualization tool provided by ROS to visualize several types of visual data). On the other hand, /CartesianPositionController uses the data for collision avoidance control, as explained later in this chapter (section 3.3).

/robot_state_publisher is an example of the benefits of the framework ROS provides for easy code reuse. This node is part of a widely used ROS package that uses the robots description (described in a .urdf file) and messages published in /joint_states to provide transformation matrices between any two joints in the robots kinematic chain. It publishes this data in /tf topics.

Figure 3.3: ROS graph



In addition, bags in ROS enable real data storage coming from sensors in the robot. These allow developers to test their algorithms with real data without having to collect new data every

time they need to run their code.

The ROS Master enables nodes to find each other and to communicate. It also holds the parameter server, which provides a central storage location for data that is relevant for several nodes. It serves, broadly speaking, as a place where global variables can be stored and retrieved by nodes. Examples of things that are stored in the parameter server are PID control parameters, the robots urdf description and frequencies at which different components living in the graph work.

Last, it should be mentioned that despite the fact that ROS is widely used in the academic context, it is lacking the reliability and robustness safety-critical systems in industrial or commercial contexts require. Some of the industrial fields, such as the automotive and the aerospace, have their own standards for the development of safety-critical software. These standards need to be certified for every software component written in any of these fields. ROS was not developed to any of these standards and it actually depends on many libraries that were not developed to these standards. ROS 2, in contrast, is based on components that are safety-certified (DDL communication) that make ROS 2 certifiable. Indeed, Apex.AI is working on Apex.OS: an API compatible to ROS 2 that is being certified ISO 26262 for safe automotive applications [12].

3.1.3 Simulation and Visualization

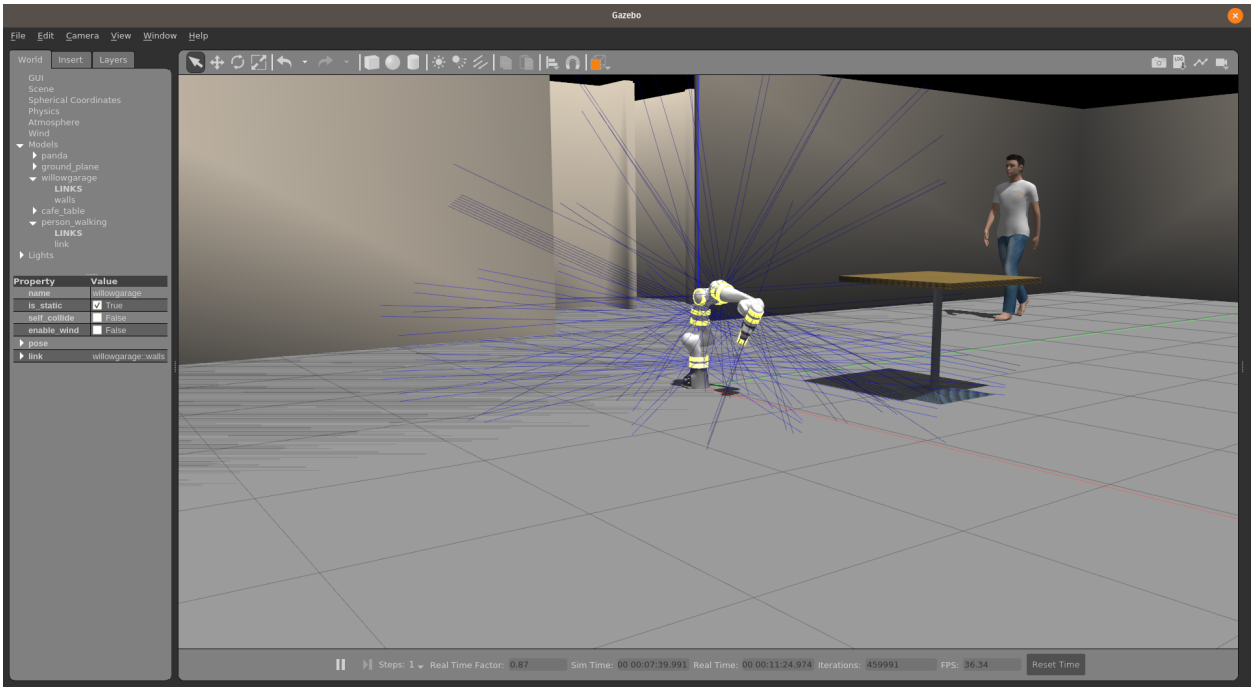
According to the Cambridge Dictionary, a simulation is a situation in which a particular set of conditions is created artificially in order to study or experience something that could exist in reality. Simulations are therefore very useful for research and especially for robotics, where the following issues are prevalent:

- Robotic hardware is usually **costy**.
- Prototyping can be **dangerous**, since bugs that appear when testing and debugging can lead to damage in the real world, as opposed to programs that only affect the virtual world.
- Development can be **slow**.

Robot simulation is a very addecuate solution to address each of these. This work has been prototyped in simulation. This has enabled parallel development of the artificial skin and its electronics on the one side and the control design on the other. The simulation software package employed has been *Gazebo*.

Gazebo [13] is a 3D robot simulator. It features dynamics simulation, advanced 3D graphics, indoor and outdoor environments and simulation of several sensors with noise that make the virtual readings realistic. It is a standalone software package, but offers integration with ROS through the *gazebo_ros_pkgs* ROS packages.

Figure 3.4: Franka Panda Emika in Gazebo's simulation environment.

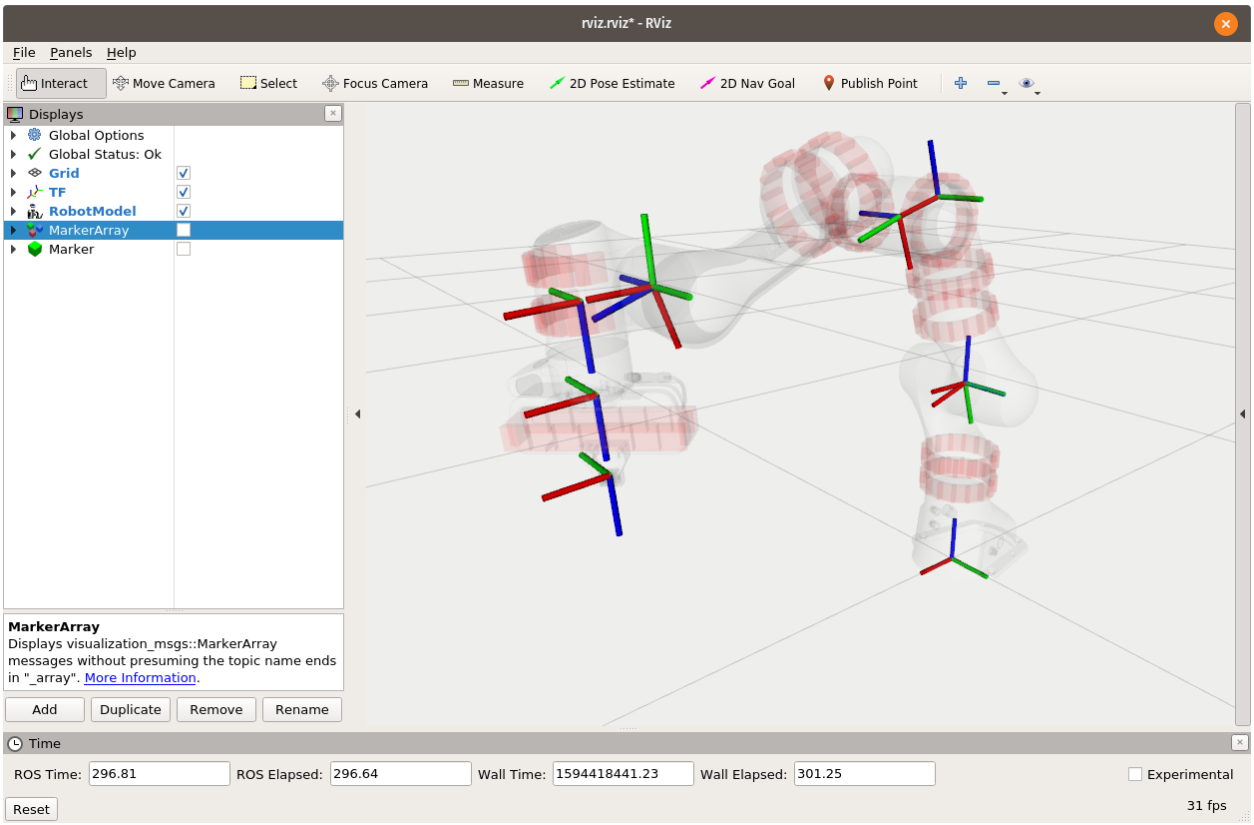


On the other hand, ROS offers a useful vizualization tool called *rviz* [14], short for *ROS Visualization*. Several kinds of data can be visualized in *rviz*, which allow to see what the robot is seeing, thinking and planning. These visualization capabilities make it an extremely useful tool for application debugging.

The difference between *Gazebo* and *rviz* is that even though both offer some kind of a graphics

visualization, *Gazebo* is mainly focused on simulating the physics whereas *rviz* is mainly useful for visualizing any kind of information concerning the robot's state and its knowledge about the environment. Note that *rviz* has the same role in both real and simulated environments.

Figure 3.5: Joint frames in Franka Panda Emika as seen in *rviz*.



3.1.4 Mathematical Optimization and Quadratic Programming

Mathematical Optimization, also called **Mathematical Programming**, is the set of principal methods used to determine the best solutions to mathematically described problems.

Formally speaking, Mathematical Optimization is the process of the formulation and solution of a optimization problem of the general mathematical form:

$$\underset{\text{w.r.t. } \mathbf{x}}{\text{minimize}}\ f(\mathbf{x}), \mathbf{x} = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^n$$

subject to the constraints:

$$g_j(\mathbf{x}) \leq 0, \quad j = 1, 2, \ldots, m$$

$$h_j(\mathbf{x}) = 0, \quad j = 1, 2, \ldots, r$$

$f(\mathbf{x})$, $g_j(\mathbf{x})$ and $h_j(\mathbf{x})$ are scalar functions of a vectorial variable $\mathbf{x}$. The components x_i of $\mathbf{x}$ are called the *variables of the optimization problem*, $f(\mathbf{x})$ is the *objective function*, $g_j(\mathbf{x})$ are the *inequality constraint functions* and $h_j(\mathbf{x})$ are the *equality constraint functions*. The optimum vector $\mathbf{x}$ that solves the optimization problem is denoted with an asterisk $\mathbf{x}^*$ and the corresponding *optimal objective function value* is $f(\mathbf{x}^*)$.

Optimization problems are usually solved using *optimization algorithms*. These optimization algorithms are tailored to particular types of optimization problems. This is why classification of optimization models is a crucial part in the optimization problem.

One important optimization problem type is the **Quadratic Programming** (QP) problem. It is an important class of problem on its own and also a subproblem for more general constrained optimization problems (sequential quadratic programming (SQP), augmented Lagrangian methods). These are the characteristics of QP problems:

- **One objective function**. The most common case is the problem being defined by a single objective function, but there are also problems that can be modeled with no objective functions or using many of them.

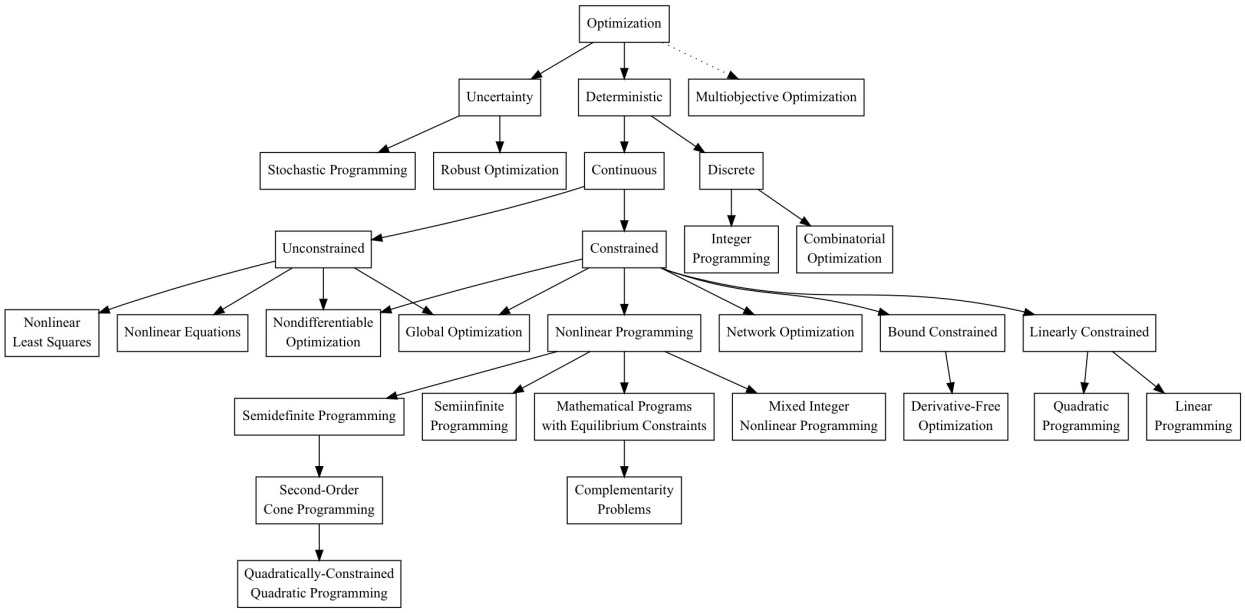
- **Deterministic optimization**. Depending on the nature of the data, an optimization problem can be either *deterministic* or *stochastic*. Due to measurement errors or more importantly because the data being used is data that represents the future, uncertainty has to be incorpored to the model. In that case we are talking about stochastic optimization. On the other hand, if data is assumed to be known with accuracy we are talking about deterministic optimization.

- **Continuous optimization problem**. For some problems, the variables of the optimization problem can only take values from a discrete set, often a subset of integers, whereas for

other problems they can take any value in $\mathbb{R}$. The former are called *discrete optimization problems* and the latter are called *continuous optimization problems*. Continuous optimization problems tend to be easier to solve because the continuity of the functions involved can be used to obtain information about the neighbourhood of the current value.

- **Constrained optimization**. Sometimes there problem does not require to specify constraints for the variables of the optimization problem. When this is the case we are talking about *unconstrained optimization*. If the problem requires a certain amount of constraints to be fulfilled it is called *constrained optimization*.

Figure 3.6: Optimization models taxonomy, with an emphasis on deterministic optimization problems.



Moreover, a problem has to meet some more requirements so that it can be written as a QP problem:

- The objective function cannot be any function, it needs to be a quadratic function that can

be rewritten as follows:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x}$$

where $\mathbf{Q}$ is a symmetric $n \times n$ matrix and $\mathbf{c} \in \mathbb{R}^{n\times 1}$.

- The constraints must be described by a linear set of equations and inequations:

$$\mathbf{A}\mathbf{x} \leq \mathbf{0}, \qquad \mathbf{A} \in \mathbb{R}^{m\times n}$$
$$\mathbf{A_{eq}}\mathbf{x} = \mathbf{0}, \quad \mathbf{A_{eq}} \in \mathbb{R}^{r\times n}$$

- Depending on the characteristics of the matrix $\mathbf{Q}$ the problem can be easily solved using polynomial-time algorithms. This is only posible if $\mathbf{Q}$ positive definite, that is, if all its eigenvalues are greater than or equal to 0. When $\mathbf{Q}$ is positive definite, the problem is a *convex QP problem*. In the contrary case, the problem is *non-convex* and only local minimums can be found.

So, the quadratic problem can be formulated as follows:

$$\begin{aligned} \text{minimize} \quad & \tfrac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{0}, \qquad \mathbf{A} \in \mathbb{R}^{m\times n} \\ & \mathbf{A_{eq}}\mathbf{x} = \mathbf{0}, \qquad \mathbf{A_{eq}} \in \mathbb{R}^{r\times n} \end{aligned} \tag{3.1}$$

3.1.5 Software Libraries

In this section we describe the software libraries chosen to fulfill the needs that have arised during the implementation of the perception and control parts of this work. *C++* is the language in which all the final code is written, so the libraries mentioned in this section are *C++ Libraries*. Other languages (*Python* and *Matlab*) have also been employed in the code prototyping phase, but as their importance is less prevalent the software components employed in those languages will be mentioned in the specific parts they have been used, in section 3.3.

3.1.5.1 Robot Kinematics

The Kinematics and Dynamics Library (KDL) [15] is a framework that provides the tools to describe robot kinematic chains and solutions to the most usual problems: forward and inverse kinematics, jacobian solvers, etc.

For that purpose, *KDL* implements classes that represent kinematic primitives such as vectors, rotations and frames as well as classes that represent the whole kinematic chain, built from segments defined by kinematic primitives. The classes that represent the whole kinematic chain are *KDL::Chain* and *KDL::Tree*, being *KDL::Tree* a container of *KDL::Chain*s. Various generic forward and inverse kinematic algorithms that take objects of these classes are provided also.

For more information on the elements mentioned in this section see section 3.3.1, where the fundamental elements for basic kinematic control are discussed.

3.1.5.2 Linear Algebra

Eigen [16] is a *template library* for linear algebra, meaning that every component in the Eigen is a *C++ template*, that is, a component whose functionality can be adapted to more than one *C++* type. In this case, Eigen supports all the *C++* standard numeric types. It is the ecosystem of *KDL* (3.1.5.1).

It supports fixed and dynamicly sized matrices and vectors and all the basic operations and common matrix decompositions. It does all its operations in an efficient, optimized, elegant and reliable fashion.

3.1.5.3 Mathematical Optimization

ALGLIB [17] is a numerical analysis and data processing library, which includes optimization algorithms among other tools. Its goal is to make high quality numerical code available to industry and academic worlds. The library is offered in a free version and in a commercial paid version, which is highly optimized. The free version, however offers access to the full library and has proved a good enough performance for this work.

3.2 Perception via proximity sensing

Proximity sensors embedded in the artificial skin provide distance information about the surroundings of the robot. These readings are one dimensional distance readings, which means that the readings of isolated individual sensors are not very useful if our objetive is to obtain information about the 3D euclidean space in which the robot's operations take place.

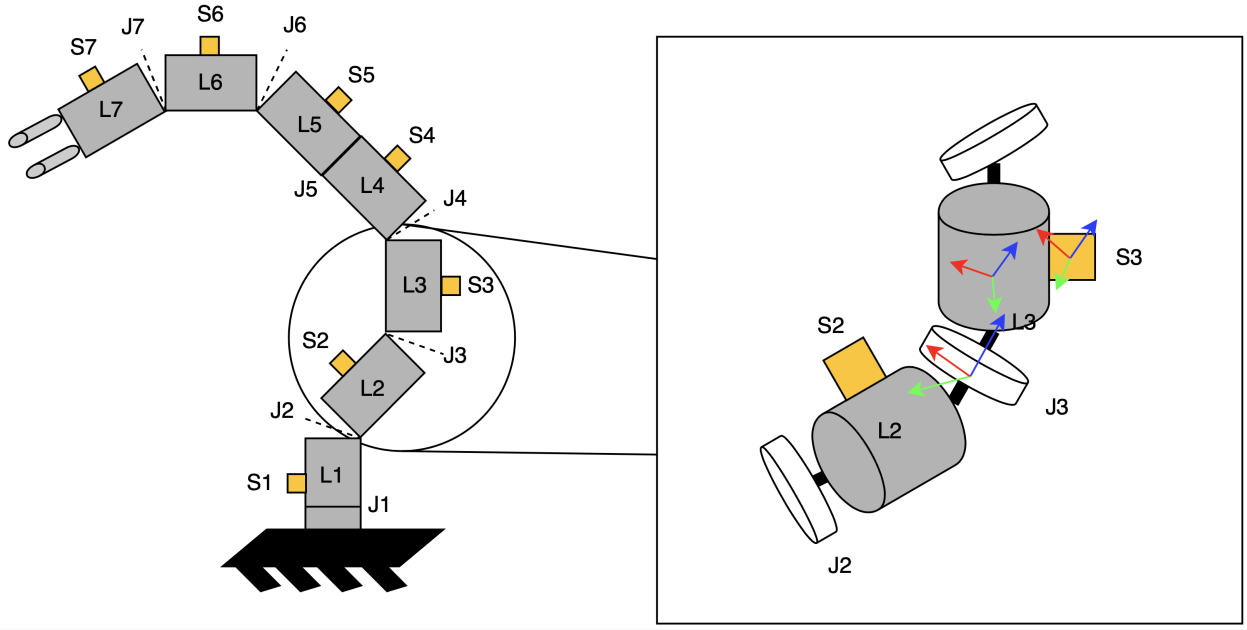
In this section we describe the method developed in this work to create a 3D point cloud from proximity distance data. The first step is the kinematic calibration of the skin units that contain the proximity sensors. Kinematic calibration makes an automatic estimation of the position and orientation of each sensor. The method is described in section 3.2.1. It is a conceptual and not detailed description, since it has been developed in a parallel project and is an essential part that make the control methods employed in this work possible. Section 3.2.3 describes the point cloud creation algorithm. Finally, section 3.2.4 shows how the obtained data is visualized in *rviz*.

3.2.1 Kinematic Calibration

This section describes a framework for automatic kinematic calibration that leverages an *Inertial Measurement Unit (IMU)* sensor to accurately estimate the position and orientation of a skin unit (SU) along the surface of a robot.

To automatically locate skin units along the surface of a robot, angular velocity and linear acceleration measurements from the IMUs are used. The position and orientation of an SU are estimated using a modified version of *Denavit-Hartenberg (DH) parameters* (see section 3.3.1.1), as illustrated in figure 3.7. The position and orientation of each SU with respect to the previous joint in the kinematic chain is estimated by six DH parameters: four parameters from the joint to a virtual joint, and two additional parameters from the virtual joint to the SU (the other two parameters are set to 0). This virtual joint is located within the link that is orthogonal to the joints z axis and the SUs z axis. This solution is necessary to adhere to the *DH* notation, so that each transformation can be expressed with no more than four parameters.

Figure 3.7: Depiction of multiple skin units (S) placed on the robots links (L) and separated by joints (J). We estimate the Denavit-Hartenberg parameters of each joint in order to calculate the pose of each skin unit along the surface of the robot.



The optimization algorithm is composed of the following **four steps**:

(1) **Initialize a kinematic chain with randomized values.** Each skin unit frame is represented using an homogeneous transformation matrix (see section 3.3.1.2).

$$^{0}T_{SU_i} = {}^{0}T_1 \cdot {}^{1}T_2 \ldots {}^{i-1}T_i \cdot {}^{i}T_{SU_i}, \quad \forall i \in \{1, 2, \ldots, n\}$$

(2) **Collect Data.** First, static forces applied to the IMU (that is, the constant acceleration due to gravity) are measured and compensated for. Then, each reference joint is moved through its operational range in a constant rotation pattern and the resulting acceleration as measured by the IMU is stored.

(3) **Define an error function.** Acceleration exerted on each SU $^{SU_i}a_{u,d}$ can be estimated as a composition of local acceleration $^{0}\mathbf{g}$, tangential acceleration $^{0}\mathbf{a}_{tan_{u,d}}$ and centripetal

acceleration ${}^{0}\mathbf{a}_{cp_{u,d}}$:

$$ {}^{0}\mathbf{a}_{tan_{u,d}} = {}^{0}\ \boldsymbol{\alpha}_d \times {}^{0}\ \mathbf{r}_{u,d} $$

$$ {}^{0}\mathbf{a}_{cp_{u,d}} = {}^{0}\ \boldsymbol{\omega}_d \times \left({}^{0}\boldsymbol{\omega}_d \times {}^{0}\ \mathbf{r}_{u,d} \right) $$

$$ {}^{SU_i}\mathbf{a}_{u,d} = {}^{SU_i}\ \mathbf{R}_0 \cdot \left({}^{0}\mathbf{g} + {}^{0}\ \mathbf{a}_{tan_u,d} + {}^{0}\ \mathbf{a}_{cp_{u,d}} \right). $$

Angular velocity ${}^{0}\boldsymbol{\omega}_d$ and angular acceleration ${}^{0}\boldsymbol{\alpha}_d$ are measured during data collection, whereas rotation matrix ${}^{SU_i}\mathbf{R}_0$ and position vector ${}^{0}\mathbf{r}_{u,d}$ can be computed using the currently estimated DH parameters.

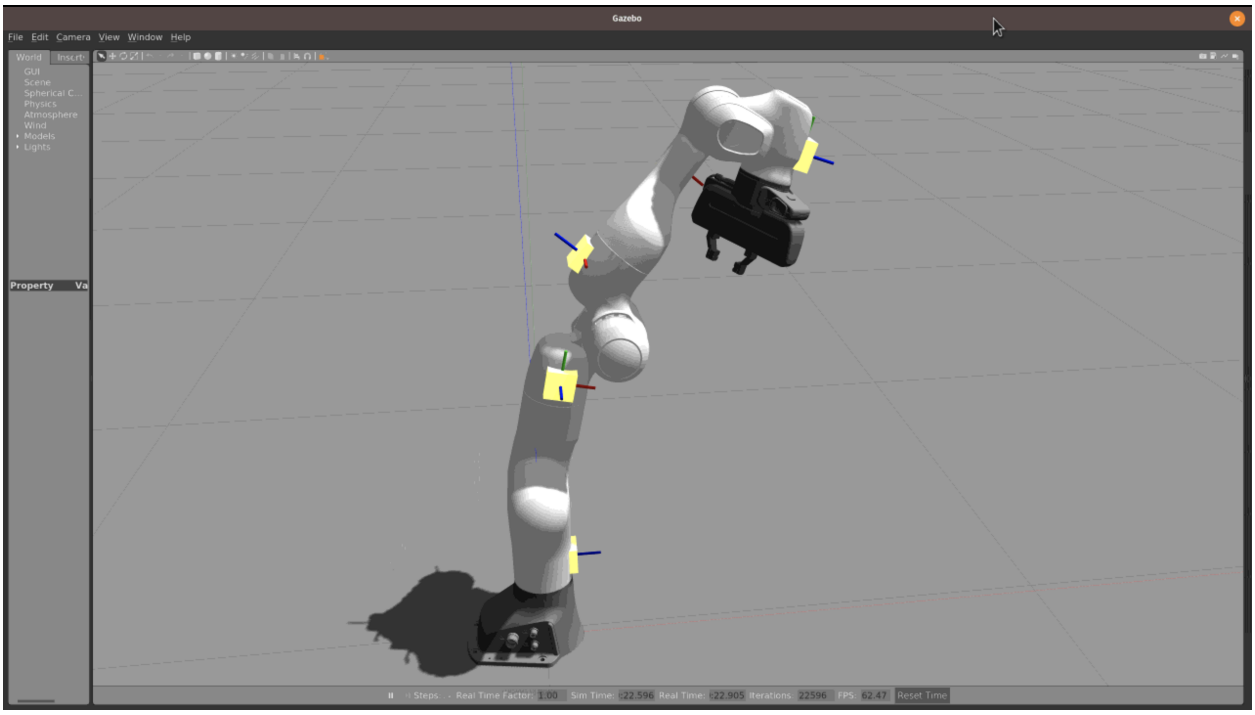
(4) **Minimize the error function with a global optimizer.** A global optimizer optimizes the DH parameters by minimizing a given error function. One example error function could be seen as the error between the measured accelerations from the IMUs and the estimated accelerations using the kinematic chain model for n_{pose} poses:

$$ E = \sum_{i=1}^{n_{pose}} \sum_{j=1}^{n_{joint}} ||a_{i,j}^{model} - a_{i,j}^{IMU}||^2 $$

Several different error functions are used to estimate both rotational and translational parameters.

The final result after calibration can be seen in figure 3.8.. Once the calibration is completed, the positions and orientation of the SU's are known. In the next section we will describe how 3D located points can be obtained from the positions and orientations of the sensors plus their readings.

Figure 3.8: Calibrated IMU positions on a simulated Franka Emika Panda robot.



3.2.2 Simulation of proximity sensors

We start by describing the algorithm's implementation with simulated sensors. There are some important differences compared to using real sensors:

- The sensors' positions and orientations are assumed to be known. A perfect outcome from calibration is assumed.
- The number of sensors that we can place on the robot's surface is unlimited.
- Sensor noise can be contolled.

In order to add proximity sensors to the robot suface in simulation *XML macros (xacro)* are utilized. *Xacro*'s allow to write shorter, easier to maintain and more readable XML files. One xacro macro expand into a larger XML expression. Moreover, it allows math, which is useful as we will see in the following lines.

We start by describing how to set up the behaviour of the proximity sensors in Gazebo. Gazebo provides the ability to add new functionalities to the *urdf* models through the use of plugins. More specificly, it is sensor plugins that provide sensor simulation functionalities. Sensor plugins are inserted in the *urdf* description of the robot model in `<gazebo>` tags. Since the sensors are meant to be attached to a link, this tag must include a reference to indicate the link in which the sensor is inserted, as seen in line 1 in the code below.

```
<gazebo reference="proximity_link${proximity_id}">                          1
    <material>Gazebo/YellowGlow</material>                                  2
    <sensor type="ray" name="proximity_sensor${proximity_id}">              3
        <pose>0 0 0 0 0 0</pose>                                            4
        <visualize>true</visualize>                                         5
        <update_rate>${freq}</update_rate>                                  6
        <ray>                                                               7
            <scan>                                                          8
                <horizontal>                                                9
                    <samples>1</samples>                                    10
                    <resolution>1</resolution>                              11
                    <min_angle>-0.0001</min_angle>                          12
                    <max_angle>0.0001</max_angle>                           13
                </horizontal>                                               14
            </scan>                                                         15
            <range>                                                         16
                <min>0.02</min>                                             17
                <max>2.0</max>                                              18
                <resolution>0.01</resolution>                               19
            </range>                                                        20
        </ray>                                                              21
        <plugin filename="libgazebo_ros_laser.so"                           22
                name="proximity_plugin${proximity_id}">                     23
            <topicName>proximity_data${proximity_id}</topicName>            24
            <bodyName>proximity_link${proximity_id}</bodyName>              25
            <updateRateHZ>${freq}</updateRateHZ>                            26
            <gaussianNoise>0.0</gaussianNoise>                              27
            <xyzOffset>0.0 0.0 0.0</xyzOffset>                              28
            <rpyOffset>0 0 0</rpyOffset>                                    29
            <frameName>proximity_link${proximity_id}</frameName>            30
        </plugin>                                                           31
    </sensor>                                                               32
</gazebo>                                                                   33
```

Listing 3.1: Setting up a ray sensor for Gazebo simulation.

Gazebo provides several sensor plugins [18], such as depth cameras, *IMU*'s or force sensors.

The proximity sensing capabilities are provided in *laser* sensors. These are originally thought to simulate *LIDAR* sensors, where a number of distance sensors are installed in one module. However, in order to simulate a single proximity sensor we only need an individual laser beam. The final configuration that accomplishes this is shown between in the `<ray>` tag.

The sensor messages are published to the topic `/proximity_data#` and the type of the messages published is `sensor_msgs/LaserScan`, which is structured as follows:

```
std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Listing 3.2: LaserScan message structure.

Once we know how to set up the sensors, we need to place them in the robotic arm. Since the arms are mainly cylindrical, cylindrical coordinate systems (r, θ, z) are a more suitable solution to refer to points on the surface than regular cartesian (x, y, z). *urdf* only allows to express positions in (x, y, z), but math provided by xacro can be utilized to express cylindrical coordinates in those required cartesian coordinates.

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \\ z &= z \end{aligned} \tag{3.2}$$

Therefore, the proximity sensor macro allows to use cylindrical coordinates to create a new link attached to the surface of the robot that simulates a new proximity sensor. The interface to the macro is defined by the arguments it takes. These is a list of these arguments:

- **proximity_id**: It is important to define a unique name for the link and joint. Trying to use an already defined name will result in an error.
- **connected_to**: It is used to select the link in which the sensor is placed. The link's coordinate frame's z axis must be oriented in the direction of the arm cylinder. In other words, link's whose coordinate frames are not oriented in this way are not valid.
- **type**: It allows to specify the joint type: fixed, revolute or prismatic. In this case, it could have been previously set to fixed.
- **x0**, **y0**: If the z axis is not centered in the desired cylinder `x0` and `y0` can apply an offset to place the origin where it is needed.
- **radius**, **theta**, **z0**: The cylindrical coordinates presented in equation 3.2.

The actual implementation of the macro can be seen in the following fragment of a *xacro* code. Cylindrical coordinates are implemented in the `<origin>` tag at line 6.

```
<!-- Connect proximity sensor to ${connected_to} -->                    1
<joint name="proximity${proximity_id}_to_${connected_to}"                 2
        type="${type}">                                                   3
    <parent link="${connected_to}"/>                                      4
    <child link="proximity_link${proximity_id}"/>                         5
    <origin                                                               6
        xyz="${x0+radius*sin(theta)} ${y0+radius*cos(theta)} ${z0}"       7
        rpy="0 0 ${pi/2 -theta}" />                                       8
</joint>                                                                  9
```

Listing 3.3: <joint> element in the proximity macro, where the position of new sensor is set up given some arguments.

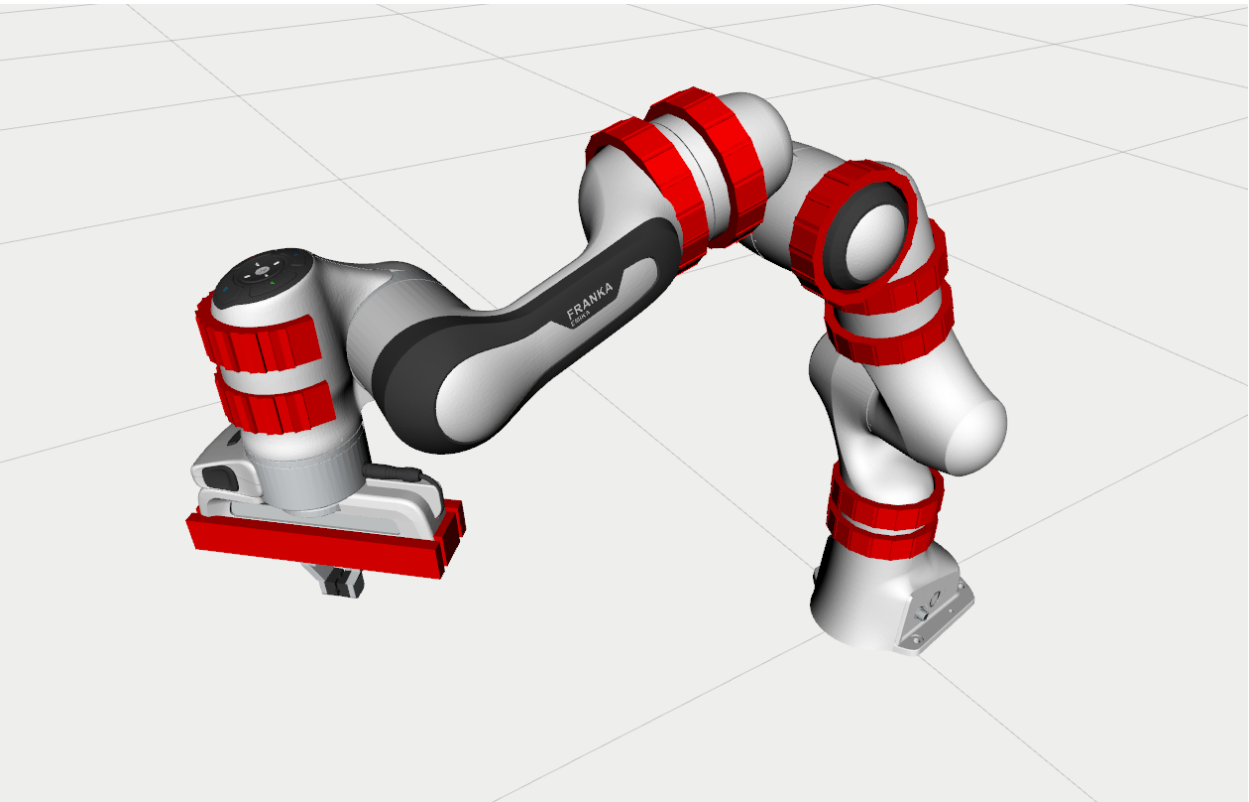
We can use this macro to place sensors through a number of rings to cover the body of the robot and simulate the artificial skin. `radius` specifies the selected link's radius and `z0` decides de longitudinal position in with we want to place the sensor. With `radius` and `z0` fixed, changing `theta` can place sensors through the entire ring `radius` and `z0` define. The following code places 4 equally spaced sensors in a ring located in panda link 3.

```
<xacro:property name="sensors_per_ring" value="4"/>
<xacro:property name="ring_id" value="0"/>
<xacro:property name="radius" value="0.06"/>
<xacro:property name="z0" value="-0.13"/>
<xacro:property name="connected_to" value="panda_link3"/>
<xacro:proximity proximity_id="${sensors_per_ring*ring_id+0}"
                 radius="${radius}"
                 theta="${2*pi/sensors_per_ring*␣0}"
                 z0="${z0}" connected_to="${connected_to}"/>
<xacro:proximity proximity_id="${sensors_per_ring*ring_id+1}"
                 radius="${radius}"
                 theta="${2*pi/sensors_per_ring*␣1}"
                 z0="${z0}" connected_to="${connected_to}"/>
<xacro:proximity proximity_id="${sensors_per_ring*ring_id+2}"
                 radius="${radius}"
                 theta="${2*pi/sensors_per_ring*␣2}"
                 z0="${z0}" connected_to="${connected_to}"/>
<xacro:proximity proximity_id="${sensors_per_ring*ring_id+3}"
                 radius="${radius}"
                 theta="${2*pi/sensors_per_ring*␣3}"
                 z0="${z0}" connected_to="${connected_to}"/>
```

Listing 3.4: Using the proximity macro to place sensors in a robot arm.

The final appearance of the new sensor links added to the surface of the robot is shown in figure 3.9, where the new links are shown as red parallelepipeds. The final amount of sensors added for simulation counts 162.

Figure 3.9: Robot body covered with proximity sensors, which appear in red.

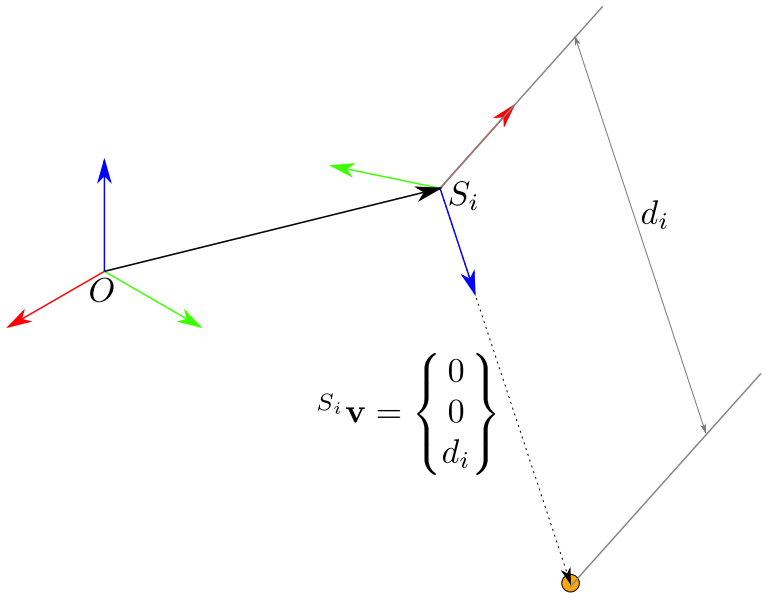


3.2.3 Point localization algorithm

The question answered in this section is: How do we convert the unidimensional reading of the proximity sensors embedded in the skin into a 3D point cloud in the cartesian space.

The diagram in figure 3.10 shows the relationship between the unidimensional reading d_i and the position vector of the corresponding point with respect to the base frame O.

Figure 3.10: Diagram showing the conversion of a proximity sensor reading into a 3D point in space.



The position of orientation ${}^{j}\mathbf{T}_{S_i}$ of the sensor S_i with respect to the joint j in which its located are obtained after the calibration process presented in section 3.2.1. This allows to obtain the position and orientation ${}^{0}\mathbf{T}_{S_i}$ of the sensor S_i with respect to the base frame O, through forward kinematics (explained in section 3.3.1.2).

$$ {}^{0}\mathbf{T}_{S_i} = {}^{0}\mathbf{T}_{j} \, . {}^{j}\mathbf{T}_{S_i} $$

Once ${}^{0}\mathbf{T}_{S_i}$ is known, the next step is calculating the position vector of obstacle associated to the sensor S_i with respect to the base frame. The distance d_i is read in the z axis of S_i (coloured in blue in the diagram in 3.10). Therefore, the position of the point detected written in the S_i coordinate frame has the following form:

$$ {}^{S_i}\mathbf{v} = \begin{Bmatrix} 0 \\ 0 \\ d_i \end{Bmatrix} $$

Again, simple forward kinematics allow us to transform that vector in order to express in in the base coordinate frame O.

$$ {}^{0}\mathbf{v} = {}^{0}\mathbf{T}_{S_i} \, {}^{S_i}\mathbf{v} $$

Expanding the expression having into account the form of homogeneous transformation matrices, ${}^{0}\mathbf{v}$ results in:

$$ {}^{0}\mathbf{v} = {}^{0}\mathbf{p}_{S_i} + {}^{0}\mathbf{R}_{S_i} \, {}^{S_i}\mathbf{v} \tag{3.3} $$

As stated in section 3.2.2, each proximity sensor publishes a `LaserScan` type message to the topic `/proximity_data#`. We have implemented a *ROS* node that deals with the transformations presented in this section when new messages are published to the topics and we have named it `proximity_listener`. The node's activity can be summarized in 3 steps:

(1) Listen to changes in the topics to which proximity sensors send data.

(2) Transform the unidimensional sensor readings into 3D position vectors.

(3) Publish the updated 3D position vectors with a certain periodicity. Also, allow saving old obstacle points for a while, so that a memory of the last reading is kept.

In the following paragraphs we expand on these steps. First, the callback function is explained, where steps (1) and (2) are carried out. Then, the publication of messages and buffer implementation are detailed in step (3).

ROS suscribers implement a callback function, which allows them to process the data published in the subscribed topics. This callback function is called every time a new message arrives to the topic. The callback that receives the proximity sensor data is presented below.

```
void sensorCallback(const sensor_msgs::LaserScan::ConstPtr& scan)
{
int sensor_number =
    std::stoi
    (
        scan->header.frame_id.substr
        (
            scan->header.frame_id.find_first_of("0123456789"),
            scan->header.frame_id.length() -1
        )
    );

try
{
    listener.lookupTransform
    (
        "/world",
        "/proximity_link" + std::to_string(sensor_number),
        ros::Time(0),
        transform[sensor_number]
    );
    translation1[sensor_number] <<
        transform[sensor_number].getOrigin().getX(),
        transform[sensor_number].getOrigin().getY(),
        transform[sensor_number].getOrigin().getZ();
    translation2[sensor_number] << 0.0,
                                   0.0,
                                   scan->ranges[0];
    rotation[sensor_number].w() =
        transform[sensor_number].getRotation().getW();
    rotation[sensor_number].x() =
        transform[sensor_number].getRotation().getX();
    rotation[sensor_number].y() =
        transform[sensor_number].getRotation().getY();
    rotation[sensor_number].z() =
        transform[sensor_number].getRotation().getZ();

    live_points[sensor_number] =
        translation1[sensor_number] +
        (rotation[sensor_number] * translation2[sensor_number]);
}
catch (tf::TransformException ex)
{
    ROS_ERROR("%s",ex.what());
    ros::Duration(1.0).sleep();
}
}
```

Listing 3.5: Point localization *C++* algorithm.

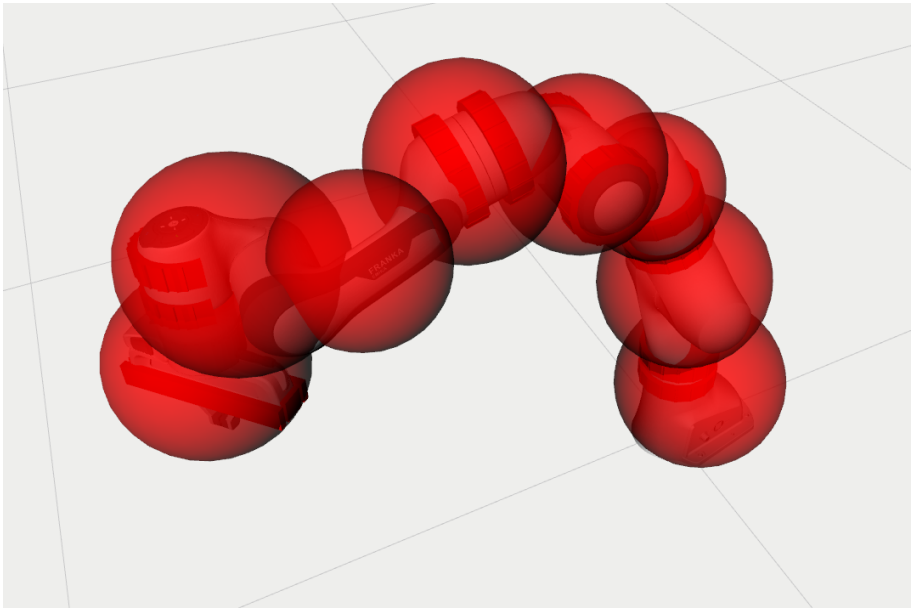
The first thing the callback function does is read the sensor from which it is receiving the message. This is done in line 3. There is a single callback for all the sensors, so this step is crucial.

Then, the sensor's position and orientation are obtained through a function that the `tf` library provides: `lookupTransform`. Also, ${}^{S_i}\mathbf{v}$ is created through the distance readings packaged in the `ranges` member of the message received. Note that as mentioned in section 3.2.2, `LaserScan` messages have the potential capability to include several sensors in one, as in *LIDAR*'s. However, in this case each sensor publishes a separate message, so `ranges` contains a unique element, indexed by `ranges[0]`.

Last, line 38 implements equation 3.3. The obtained vector is saved in the position corresponding to the sensor that is sending the message of the `std::vector live_points`.

As far as the publication of this processed data is concerned, there are two things that need to be taken into account. On the one hand, sensed points that come from the robot itself's need to be removed. We do this by covering the whole robot body with a set of spheres, shown in figure 3.11, and checking if the points are inside those spheres. This is done in `isInSphere` in line 10 of listing 3.6.

Figure 3.11: Sphere coverage of the robot for removal of the points detected its the surface.



```
geometry_msgs::Point point;
ros_robotic_skin::PointArray msg;

ros::Rate rate(50.0);
while (ros::ok())
{
    for (int i = 0; i < num_sensors; i++)
    {
        if (std::isnan(live_points[i].x()) ||
            isInSphere(live_points[i]) ||
            (live_points[i].z() < floor_threshold) && removeFloor);
        else
        {
            point.x = live_points[i].x();
            point.y = live_points[i].y();
            point.z = live_points[i].z();
            msg.points.push_back(point);
        }
    }
    pub.publish<ros_robotic_skin::PointArray>(msg);
    msg.points.clear();
    rate.sleep();
}
```

Listing 3.6: Body and floor points removal and publication *C++* algorithm.

Another optional filter that can be applied to the perveived points is that if the boolean variable `removeFloor` is set to true, points that are lower than a certain threshold `floor_threshold)`, in meters, will also be removed. Also, if a sensor i does not perceive any point in the range for which it is set up, `live_point[i]` will be

$$\begin{bmatrix} \text{NaN} & \text{NaN} & \text{NaN} \end{bmatrix}^{\intercal}$$

and the point will not be added to the message.

The points are packed in a custom created message called `ros_robotic_skin::PointArray` , whose only field is an array of `geometry_msgs::Point`s. This is done at 100 Hz, or what is the same, every 10 ms.

Moreover, the points obtained at each of the iterations are saved in a control buffer. This is the portion of the proximity listener that implements this behaviour, where `buffer` is a data structure of type `std::vector<Eigen::Vector3d>`.

```
if (buffer.size() == 10)                                    1
{                                                           2
    buffer.pop_back();                                      3
}                                                           4
buffer.insert(buffer.begin(), final_points);                5
```

Listing 3.7: Initializer of the ProximityVisualizer class

3.2.4 Visualization in *rviz*

In this section we extoplain the last step of the perception part: visualization. Visualization of the data perveived by the robot is a very intuitive way of seeing what is going on, both on the perception and control algorithms. We will see how to visualize the data processed by the `proximity_listener` node in *rviz*.

There are several dysplay types that can be visualized in rviz. We use `Marker Array`s [19] to visualize the point cloud generated.

We have called the class that implements the visualization `ProximityVisualizer`. In the

initializer of the class, shown in listing 3.8, the node is suscribed to the data published in the topic `live_points` by `proximity_listener`. The data received in that topic will be processed in `ProximityVisualizer::Callback`. Next, the publication of messages of type `MarkerArray` in the topic `visualization_marker_array` is set up. The general properties that are kept through all of the `Marker`s that will be added to the `MarkerArray` are also set up and the rest of the properties are initialized.

Note that these properties allow to set up the `Marker`'s shape, the size, the color, the coordinates in which it will be located, as well as the frame in which those coordinates are written and the orientation. `namespace` allows organization within the same topic so that `Marker`s can be further classified. The `id` of the marker is unique and whenever a new marker with the same `id` is published, the previous one is deleted. Finally, `action` can be `ADD`, `DELETE` or `DELETEALL`.

```
ProximityVisualizer::ProximityVisualizer()
{

    sub = n.subscribe<ros_robotic_skin::PointArray>
        ("live_points", 1, &ProximityVisualizer::Callback, this);
    pub = n.advertise<visualization_msgs::MarkerArray>
        ("visualization_marker_array", 1);
    // Marker properties that are the same for all the points
    marker.ns = "Live Points";
    marker.action = visualization_msgs::Marker::ADD;
    marker.scale.x = 0.05;
    marker.scale.y = 0.05;
    marker.scale.z = 0.05;
    marker.color.r = 1.0;
    marker.color.g = 0.0;
    marker.color.b = 0.0;
    marker.color.a = 1.0;
    marker.header.frame_id = "/world";
    marker.type = visualization_msgs::Marker::SPHERE;
    marker.pose.orientation.x = 0.0;
    marker.pose.orientation.y = 0.0;
    marker.pose.orientation.z = 0.0;
    marker.pose.orientation.w = 1.0;
    // Marker properties that depend on the point
    marker.header.stamp = ros::Time::now();
    marker.id = 0;
    marker.pose.position.x = 0.0;
    marker.pose.position.y = 0.0;
```

```
    marker.pose.position.z = 0.0;
}
```

Listing 3.8: Initializer of the ProximityVisualizer class

Every time a new message arrives to the `live_points` topic in runtime, the callback will add the new points received to a `MarkerArray`. This is shown in listing 3.9.

The sequence when a new message arrives is as follows. First, all the previously published markers are deleted publishing a `MarkerArray` that contains a `Marker` whose `action` is `DELETEALL`. Then, the new points received are packed into a `Marker` and pushed back in a `MarkerArray`. Finally, this `MarkerArray` is published.

```
    void ProximityVisualizer::Callback(
        const ros_robotic_skin::PointArray::ConstPtr& msg)
    {
        // Delete all points from previous callback
        marker.action = visualization_msgs::Marker::DELETEALL;
        marker_array.markers.push_back(marker);
        pub.publish<visualization_msgs::MarkerArray>(marker_array);
        marker_array.markers.clear();

        // Add new points
        marker.action = visualization_msgs::Marker::ADD;
        marker.header.stamp = ros::Time::now();
        for (int i = 0; i < msg->points.size(); i++)
        {
            // Check that it's not 'nan' or 'inf'
            marker.id = i;
            marker.pose.position.x = msg->points[i].x;
            marker.pose.position.y = msg->points[i].y;
            marker.pose.position.z = msg->points[i].z;
            marker_array.markers.push_back(marker);
        }
        pub.publish<visualization_msgs::MarkerArray>(marker_array);
        marker_array.markers.clear();
    }
```

Listing 3.9: Callback of the ProximityVisualizer class

In order to visualize the `MarkerArray`, a subscription to the topic has to be added in *rviz*, as shown in figure 3.12. Since the `Namespaces` field is empty, all the points in the received `MarkerArray` will be visualized, with no exception.

Figure 3.12: Menu of the Marker Array plugin in *rviz*.

3.3 Control

This is the main section of this work. Here we present all the concepts necessary for safe motion control, from fundamental robot kinematics to obstacle avoidance algorithms and different means by which robot redundancy can be leveraged.

3.3.1 The Kinematic Control Problem

In this section one of the most fundamental problems in robotics will be defined and solved: the Kinematic Control Problem. It can be formulated as the problem of obtaining a joint space trajectory $\mathbf{q}(t)$ that satisfies a desired trajectory in the cartesian space $\mathbf{x}(t)$.

This problem contains several elements required in order to solve the collision avoidance problem in subsequent sections. Moreover, the collision avoidance will be built as a modification to the more fundamental method described in this section.

3.3.1.1 Geometric representation of the kinematic chain of a robot

Kinematics is the branch of physics that studies the motion of the bodies. It does that with no consideration of the causes that originate that motion, forces and torques. Kinematics is required and fundamental in robotics for design, analysis, control and simulation.

In order to study the kinematics of a robot its geometric representation must be defined mathematically beforehand. Rigid body kinematics deals with the problem of studying relative movement of two non-deformable elements by attaching a coordinate frame to each of them. Robot

manipulators and robots in general can be seen as a series of rigid links connected by joints. Therefore, attaching a frame to each of the links of a robot in a convenient manner is how robots' geometry is usually represented.

However, there are infinitely many ways in which this 3-dimensional frames can be assigned to the links. The origin and rotation of the frames could be located anywhere in the space as long as its movement was fixed to the respective links.

That is where conventions for geometric representation come in place. The most well known and extended convention was introduced by Denavit and Hartenberg (DH) in 1955 [20]. Although there have been numerous adaptations, the representations serve the same purpose and DH parameters are still widely used. In fact, the manufacturer of the robot employed in this work (3.1.1) provides DH parameters in its documentation [10].

DH suggest a set of rules to place the frames corresponding to each robot link:

- The n links of the mechanism are numbered from 0 to N, being link 0 the fixed base.
- The n joints of the mechanism are numbered from 1 to N, with joint i located between links $i-1$ and i.
- The $\mathbf{z_i}$ axis is located along the axis of joint i.
- The $\mathbf{x_{i-1}}$ axis is located along the common perpendicular between $\mathbf{z_{i-1}}$ and $\mathbf{z_i}$ axis.

Then, they define 4 parameters that completely define the position and rotation of a link with respect to the previous one. Note that full position and orientation are usually defined by at least 6 parameters. However, geometric constraints inherent to robotic arms allow us to define them with just 4 parameters.

The four parameters are:

- $\mathbf{a_i}$: the distance from $\mathbf{z_{i-1}}$ to $\mathbf{z_i}$ along $\mathbf{x_{i-1}}$.
- $\alpha_\mathbf{i}$: the angle from $\mathbf{z_{i-1}}$ to $\mathbf{z_i}$ about $\mathbf{x_{i-1}}$.

- $\mathbf{d_i}$: the distance from $\mathbf{x_{i-1}}$ to $\mathbf{x_i}$ along $\mathbf{z_i}$.
- $\theta_\mathbf{i}$: the angle from $\mathbf{x_{i-1}}$ to $\mathbf{x_i}$ about $\mathbf{z_i}$.

Most common joint types in robot manipulators are revolute and prismatic joints. In the case of **revolute joints** all the parameters but $\theta_\mathbf{i}$ are defined by the geometric design of the robot, while $\theta_\mathbf{i}$ is the variable of the movement. If the joint is **prismatic**, all the parameters but $\mathbf{d_i}$ are defined by the geometry of the robot, while $\mathbf{d_i}$ is the variable of the movement.

Franka Panda Emika (described in section 3.1.1) is equipped with 7 revolute joints. Its DH parameters are given by the manufacturer and shown in figure 3.1.

In ROS, the robots' geometry is internally described in *urdf* files. *urdf* files make use of *Extensible Markup Language (XML)*, the same markup language that is used for web pages in *html* files. The following fragment taken from the complete descripion of Franka Panda (by justagist [21]) Emika describes the transformation from frame 6 to frame 7.

```
<joint name="${arm_id}_joint7" type="revolute">                          1
      <safety_controller k_position="100.0"                              2
                         k_velocity="40.0"                               3
                         soft_lower_limit="-2.8973"                      4
                         soft_upper_limit="2.8973"/>                     5
      <origin rpy="${pi/2} 0 0" xyz="0.088 0 0"/>                        6
      <parent link="${arm_id}_link6"/>                                   7
      <child link="${arm_id}_link7"/>                                    8
      <axis xyz="0 0 1"/>                                                9
      <limit effort="12" lower="-2.8973"                                 10
                         upper="2.8973"                                  11
                         velocity="2.6100"/>                             12
      <dynamics damping="1.0" friction="0.5"/>                           13
</joint>                                                                 14
```

Listing 3.10: Fragment of panda_arm.urdf

The relative positions and orientations are defined under the tag `<origin/>` in line 6. DH parameters are not used in this tag, even though *urdf* could be adapted to use them through the language *xacro*, that permits to do math inside textiturdf. Rotation is defined through *Euler Angles*: rotations are defined by applying rotation about x, y and z axes, in this order. Translation is defined with a 3×1 vector that goes from frame $i-1$ to frame i, from 6 to 7 in this case.

The use of DH parameters would have been beneficial and less error prone through the process of writing the *urdf* description.

In the next section we will make use of the geometric representation of the robot to locate the diferent elements of the robot.

3.3.1.2 Forward Kinematics

Forward kinematics provides a mathematical tool that allows to find the position and orientation of the end effector or any other point in the kinematic chain given all the joint variables' values. These are ($\theta_\mathbf{i}$s for revolute joints and $\mathbf{d_i}$s for prismatic joints.

A common way to represent the rotation and translation from one axis to another are homogeneous transformation matrices. They are 4×4 matrices defined as in equation 3.4.

$$ {}^{i-1}\mathbf{T}_i = \begin{pmatrix} {}^{i-1}\mathbf{R}_i & {}^{i-1}\boldsymbol{p}_i \\ \mathbf{0}^{\mathrm{T}} & 1 \end{pmatrix} \tag{3.4} $$

The top left 3×3 matrix ${}^{i-1}\mathbf{R}_i$ defines the rotation from axis $i-1$ to axis i. ${}^{i-1}\mathbf{R}_i$ is an orthogonal matrix, this meaning that the basis formed by its columns is orthonormal. Each of the column vectors are required to have unit length. One nice property of orthogonal matrices is that their inverse is exactly the same as their transpose:

$$ ({}^{i-1}\mathbf{R}_i)^{-1} = {}^{i}\,\mathbf{R}_{i-1} = ({}^{i-1}\mathbf{R}_i)^{\intercal} $$

On the other hand, the top right 3×1 position vector ${}^{i-1}\boldsymbol{p}_i$ defines the translation vector from axis $i-1$ to axis i.

Homogeneous transformation matrices are an easy and convenient way to calculate several linked transformations. ${}^{i-2}\mathbf{T}_i$ would be calculated as follows:

$$ {}^{i-2}\mathbf{T}_i = {}^{i-2}\,\mathbf{T}^{i-1}_{i-1}\mathbf{T}_i = \begin{pmatrix} {}^{i-2}\mathbf{R}_{i-1} & {}^{i-2}\boldsymbol{p}_{i-1} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{pmatrix} \begin{pmatrix} {}^{i-1}\mathbf{R}_i & {}^{i-1}\boldsymbol{p}_i \\ \mathbf{0}^{\mathrm{T}} & 1 \end{pmatrix} = \begin{pmatrix} {}^{i-2}\mathbf{R}^{i-1}_{i-1}\mathbf{R}_i & ({}^{i-2}\boldsymbol{p}_{i-1} + {}^{i-2}\,\mathbf{R}^{i-1}_{i-1}\boldsymbol{p}_i) \\ \mathbf{0}^{\mathrm{T}} & 1 \end{pmatrix} $$

This is the desired behaviour because ${}^{i-2}\mathbf{R}_{i-1}^{i-1}\mathbf{R}_i$ represents the combination of applying the rotation ${}^{i-2}\mathbf{R}_{i-1}$ first and ${}^{i-1}\mathbf{R}_i$ after. $({}^{i-2}\boldsymbol{p}_{i-1} + {}^{i-2}\,\mathbf{R}_{i-1}^{i-1}\boldsymbol{p}_i)$ takes the vector ${}^{i-1}\boldsymbol{p}_i$ to the $i-2$ basis and them it sums the origin of the frame $i-1$, ${}^{i-2}\boldsymbol{p}_{i-1}$, already written in the $i-2$ axis.

The homogeneous transformation matrix ${}^{i-1}\mathbf{T}_i$ can be built from the DH parameters representation with $\mathbf{a_i}$, $\alpha_\mathbf{i}$, $\mathbf{d_i}$ and $\theta_\mathbf{i}$. For that purpose the following steps are combined in order:

(1) $\alpha_\mathbf{i}$ rotation about the axis x_{i-1}.

(2) $\mathbf{a_i}$ translation through the axis x_{i-1}.

(3) $\theta_\mathbf{i}$ rotation about the axis z_i.

(4) $\mathbf{d_i}$ translation about the axis z_i.

$$
{}^{i-1}\mathbf{T}_i = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

$$
= \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_i \\ \sin\theta_i\cos\alpha_i & \cos\theta_i\cos\alpha_i & -\sin\alpha_i & -\sin\alpha_i d_i \\ \sin\theta_i\sin\alpha_i & \cos\theta_i\sin\alpha_i & \cos\alpha_i & \cos\alpha_i d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

Once all the the transformation matrices ${}^{i-1}\mathbf{T}_i$ are defined for $i = 1, \ldots, n$, the full transformation from the base $i = 0$ to the end effector $i = n$ transformation can easily be calculated:

$$
{}^0\mathbf{T}_n = {}^0\,\mathbf{T}_1^1\mathbf{T}_2 \ldots {}^{n-1}\,\mathbf{T}_n
$$

Forward Kinematics are calculated in the robot_state_publisher package [22]. The robot's description is read from a *urdf* file loaded in the parameter server. Then, the joints' values are read

from a topic named */joint_states* and used to compute forward kinematics of all the joints. The results are published in a topic named */tf*.

3.3.1.3 Inverse Kinematics

The inverse Kinematics problem is the opposite to the one of Forward Kinematics (section 3.3.1.2). The objective in this case is not to locate the robot's parts once known joint values, but to find the joint values take the end effector (or any other point in the kinematic chain) to a given position and orientation.

Given the nature of the homogeneous transformation matrices defined in section 3.3.1.2, whose elements contain sin and cos functions, the problem requires to solve a set of non linear equations defined by:

$$ {}^0\mathbf{T}_n(q_1, \ldots, q_n) = ({}^0\mathbf{T}_n)^d $$

It is trivial to see that solving the position ${}^0\boldsymbol{p}_n(q_1, \ldots, q_n) = ({}^0\boldsymbol{p}_n)^d$ describes 3 equations. On the other hand, since rotation matrices provide six auxiliary relationships (3 coming from the fact that column vectors to have unit length, and 3 coming from the requirement of its column vectors being mutually orthogonal), and they contain 9 elements, $9 - 6 = 3$ more equations are coming from ${}^0\mathbf{R}_n(q_1, \ldots, q_n) = ({}^0\mathbf{R}_n)^d$.

In order a solution to exist, the desired location and rotation $({}^0\mathbf{T}_n)^d$ needs to be in the robot's workspace.

A closed form of the equation does not always exist. In fact, in the common case there is no way to do this. Consequently, numerical methods are needed.

For Franka Panda Emika $n = 7$, there are infinitely many joint values that satisfy the Inverse Kinematics equations.

3.3.1.4 Forward Instantaneous Kinematics

Forward instataneous Kinematics relates the motion rates of joints $\mathbf{q} = \begin{bmatrix} q_1 & \dots & q_n \end{bmatrix}^\intercal$ with the velocities in the 3D Cartesian Space of the end effector.

Note that everything defined for the specific point in the kinematic chain *end effector* will also be adaptable to any other point in the kinematic chain, by simply considering a reduced kinematic chain that goes from the base to joint i, instead of to the end effector.

We start by redefining what we got in section 3.3.1.2. This was our final result, given in the form of homogeneous transformation matrices:

$$ {}^0\mathbf{T}_n = {}^0\,\mathbf{T}_1^1\mathbf{T}_2 \dots {}^{n-1}\,\mathbf{T}_n = \begin{pmatrix} {}^0\mathbf{R}_n & {}^0\boldsymbol{p}_n \\ \mathbf{0}^{\mathrm{T}} & 1 \end{pmatrix} $$

The position of the end effector is well defined by the position vector ${}^0\boldsymbol{p}_n$. However, the rotation matrix ${}^0\boldsymbol{p}_n$, by definition of rotation matrix, contains six auxiliary relationships (3.3.1.2). In consequence, rotation matrices are not minimal representations. For Forward Instantaneous Kinematics we will use a minimal representation of orientation, such as Euler angles (3.3.1.1), where orientation is completely described with just three elements.

$$ \mathbf{t}(t) = \mathbf{f}(t) = \begin{bmatrix} f_1(t) \\ f_2(t) \\ f_3(t) \\ f_4(t) \\ f_5(t) \\ f_6(t) \end{bmatrix} $$

$\mathbf{f}$ is a differentiable nonlinear vector function. $f_1(t)$, $f_2(t)$ and $f_3(t)$ describe the position of the end effector and $f_4(t)$ $f_5(t)$ and $f_6(t)$ its orientation.

Thus, the rate of change of this parameters is given by $\dot{\mathbf{t}}(t)$,

$$\dot{t}_i(t) = \frac{dt_i(t)}{dt} = \sum_{i=1}^{n} \frac{\partial t_i}{\partial q_i} \cdot \frac{dq_i}{dt} = (\nabla_{\mathbf{q}} t_i)^{\intercal} \cdot \dot{\mathbf{q}}$$

In matrix form,

$$\dot{\mathbf{t}} = \frac{d\mathbf{t}}{dt} = [\frac{\partial \mathbf{t}}{\partial \mathbf{q}}] \cdot \dot{\mathbf{q}} = \mathbf{J_t}(\mathbf{q}) \cdot \dot{\mathbf{q}}$$

$\mathbf{J_t}(\mathbf{q}) \in \mathbb{R}^{6\times n}$ is called the *Jacobian*. It relates joint space velocities to cartesian space velocities and it is a function of the joint angles (or distances in the case of prismatic joints) $\mathbf{q}$. Its ith row is the gradient of the component i in $\mathbf{t}(t)$ $\nabla_{\mathbf{q}} t_i$.

It is convenient to remark that, with this formulation, the components of the velocity vector $\dot{\mathbf{t}}(t)$ express the rate of change of the parameters that form the minimal representation vector $\mathbf{t}(t)$. In Rigid Body Kinematics, the velocities of all the points in a rigid body (a link in our case) are completely defined by two vectors [23].

(1) The *angular velocity* of the body $\boldsymbol{\omega}$.

(2) The velocity of one point pertaining to the body $\mathbf{v}_O$.

Given those two vectors the velocity $\mathbf{v}_P$ of any other point P pertaining to the same body can be calculated as:

$$\mathbf{v}_P = \mathbf{v}_O + \boldsymbol{\omega} \times \overrightarrow{\mathbf{OP}} \tag{3.5}$$

However, while the end effector velocity represented by the first three elements of $\dot{\mathbf{t}}(t)$ is suitable for $\mathbf{v}_O$, the last three elements of $\dot{\mathbf{t}}(t)$ don't express the angular velocity $\boldsymbol{\omega}$ of the end effector link. They represent the rate of change of Euler Angles or any other minimal representation parameters chosen to describe orientation.

The actual Jacobian that relates the joint space velocities to $\dot{\mathbf{x}} = \begin{bmatrix} \mathbf{v}_{\text{end effector}} \\ \boldsymbol{\omega}_{\text{end effector}} \end{bmatrix}$ is:

$$\mathbf{J}(\mathbf{q}) = \mathbf{T}(t) \cdot \mathbf{J_t}(\mathbf{q})$$

$$\mathbf{T}(t) = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{R}(t) \end{bmatrix}$$

$\mathbf{T}(t)$ only depends on time t and its expression depends on the minimal representation chosen to describe orientation. The resulting equation is:

$$\dot{\mathbf{x}} = \mathbf{J}(\mathbf{q}) \cdot \dot{\mathbf{q}} \tag{3.6}$$

There are many methods that compute the values of the jacobian matrix $\mathbf{J}(\mathbf{q})$ [24]. The general idea underlying is similar in all of them, with differences in efficiency.

$$\mathbf{J}(\mathbf{q}) = \begin{bmatrix} \boldsymbol{\mathcal{J}}_{P1} & \boldsymbol{\mathcal{J}}_{P2} & \dots & \boldsymbol{\mathcal{J}}_{Pn} \\ \boldsymbol{\mathcal{J}}_{O1} & \boldsymbol{\mathcal{J}}_{O2} & \dots & \boldsymbol{\mathcal{J}}_{On} \end{bmatrix}$$

$\boldsymbol{\mathcal{J}}_{Pi} \in \mathbb{R}^{3\times 1}$ denotes the contribution of $\dot{q}_i$ per radian to $\mathbf{v}_{\text{end effector}}$. $\boldsymbol{\mathcal{J}}_{Oi} \in \mathbb{R}^{3\times 1}$ denotes the contribution of $\dot{q}_i$ per radian to $\boldsymbol{\omega}_{\text{end effector}}$. Their value is computed as follows:

$$\boldsymbol{J}_{Pi} = \begin{cases} {}^0\mathbf{z}_{i-1} & \text{if joint } i \text{ is prismatic} \\ {}^0\mathbf{z}_{i-1} \times^{i-1} \boldsymbol{p}_{\text{end effector}} & \text{if joint } i \text{ is revolute} \end{cases}$$

$$\boldsymbol{J}_{Oi} = \begin{cases} \mathbf{0} & \text{if joint } i \text{ is prismatic} \\ {}^0\mathbf{z}_{i-1} & \text{if joint } i \text{ is revolute} \end{cases}$$

As described in section 3.3.1.1, ${}^0\mathbf{z}_{i-1}$ is joint i's z axis, which is a unitary vector in the direction of the joint. ${}^0\mathbf{z}_{i-1}$ coincides both the direction in which a prismatic joint contributes to the end effector velocity $\mathbf{v}_{\text{end effector}}$ and with the contribution of a revolute joint to the angular velocity of the end effector $\boldsymbol{\omega}_{\text{end effector}}$. The magnitude $\mathbf{z}_{i-1} \times^{i-1} \boldsymbol{p}_{\text{end effector}}$ is due to the $\boldsymbol{\omega} \times \overrightarrow{\mathbf{OP}}$ part in equation 3.5. It is obvious that the contribution of a prismatic joint to $\boldsymbol{\omega}_{\text{end effector}}$ is $\mathbf{0}$.

In section 3.1.5.1 we presented KDL as the library we would use to deal with problems dealing with fundamental robot kinematics. This is how the robotic jacobian is obtained in KDL:

```
Eigen::MatrixXd KDLSolver::computeJacobian
(
    std::string linkName,
    Eigen::VectorXd q
)
{
    // Obtain the description of the robot up to link linkName
    KDL::Chain kdlChain;
    kdlTree.getChain("panda_link0", linkName, kdlChain);
    int number_joints = kdlChain.getNrOfJoints();
    // Construct the jacobian solver with the description obtained
    KDL::ChainJntToJacSolver JSolver =
        KDL::ChainJntToJacSolver(kdlChain);
    // Prepare the data for the solver
    KDL::Jacobian J;
    J.resize(number_joints);
    KDL::JntArray KDLJointArray(7);
    KDLJointArray.data = q;
    KDLJointArray.resize(number_joints);
    // Obtain the jacobian
    JSolver.JntToJac(KDLJointArray, J);
    return J.data;
}
```

Listing 3.11: Implementation of the function `computeJacobian`.

Note that `linkName` lets us decide which is the link for we want to obtain the jacobian. If the link is other than the end effector we will call it *partial jacobian*.

Later, in section 3.3.3 we will upgrade this function so that it will let us obtain the jacobian for any point attached to a link and not only the one corresponding to the joints.

3.3.1.5 Inverse Instantaneous Kinematics

As defined at the beginning of 3.3.1, the kinematic control problem deals with the problem of obtaining a joint space trajectory $\mathbf{q}(t)$ that satisfies a desired trajectory in the cartesian space $\mathbf{x}(t)$. Several approaches can be taken in order to find a solution to this problem, but according to [25], the most relevant ones for redundant manipulators (sections 3.1.1 and 3.3.1.2) solve the problem at the joint velocity level.

Therefore, joint space velocities $\dot{\mathbf{q}}(t)$ that result in end effector velocities $\dot{\mathbf{x}}_\mathbf{d}$ that satisfy a desired trajectory in the cartesian space $\mathbf{x}_\mathbf{d}(t)$ must be found. This is the opposite problem to the

one presented in the previous section (3.3.1.4) that was resolved in equation 3.6. The equation presents a linear system of equations in the unknowns $\dot{\mathbf{q}}$. This contrasts with the fact that, as stated in section 3.3.1.3, the Inverse Kinematics problem is non-linear. The jacobian $\mathbf{J}(\mathbf{q})$ is a local linearization of the instantaneous problem, which is valid only for the specific time instant t and joint values $\mathbf{q}$ for which it was calculated. In the general task that than be identified by m variables:

$$\underset{m\times n}{\mathbf{J}(\mathbf{q})} \cdot \underset{n\times 1}{\dot{\mathbf{q}}} = \underset{m\times 1}{\dot{\mathbf{x}}_{\mathbf{d}}} \tag{3.7}$$

It is important to note that the maximum number of variables a task can be identified with is $m = 6$. This implies that for a redundant robot for which $n >= 7$, $m > n$ will always remain true.

In the following paragraphs we will utilize some definitions and theorems of Linear Algebra to verify that the linear system of equations defined by equation 3.7 has a set of solutions that can only be either empty or infinite. We also verify that when the $\mathbf{J}(\mathbf{q})$ is full rank, i.e. when its rank is exactly m, the vector space of solutions has dimension $(m - n)$. When the rank is $< m$, there exists no solution.

We start by listing some definitions and theorems of Linear Algebra (taken from [26]. However this are well known results and definitions and can be found in any general linear algebra book).

Definition 1. *The **row space/column space** of a matrix is the span of the set of its rows/columns. The **row rank/column rank** is the dimension of this space, the number of linearly independent rows/columns.*

Theorem 2. *For any matrix, the row rank and column rank are equal.*

Definition 3. *The **rank** of a matrix is its row rank or its column rank.*

Theorem 4. *No linearly independent set can have a size greater than the dimension of the enclosing space.*

Therefore, because of this last theorem, it is obvious that the maximum rank for a given matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the minimum value between m and n. The rank of this matrix can only be equal to or less than that maximum rank. In the case of redundant robots, the maximum rank $\mathbf{J}(\mathbf{q})$ will always be given by the dimension of the row space m, that is, the number of variables needed to define the task.

Theorem 5. *Any linear system's solution set has the form*

$$\left\{\vec{p} + c_1\vec{\beta}_1 + \cdots + c_k\vec{\beta}_k \mid c_1, \ldots, c_k \in \mathbb{R}\right\}$$

where $\vec{p}$ is any particular solution and where the number of vectors $\vec{\beta}_1, \ldots, \vec{\beta}_k$ equals the dimension of the space of solutions associated to homogeneous system.

Theorem 6. *For linear systems with n unknowns and with matrix of coefficients* $\mathbf{A}$*, the statements*

(1) the rank of $\mathbf{A}$ *is r*

(2) the vector space of solutions of the associated homogeneous system has dimension (n-r)

are equivalent.

Consequently, the homogeneous system associated to equation 3.7, $\mathbf{J}(\mathbf{q})\cdot\dot{\mathbf{q}} = \mathbf{0}$, has a solution space of dimension $(\mathrm{n} - \mathrm{r}) \geq (\mathrm{n} - \max(\mathrm{r})) = (\mathrm{n} - \mathrm{m})$. In the case of Panda Franka Emika, the solution space of the associated homogeneous system will always be greater than or equal to 1: $(\mathrm{n} - \mathrm{r}) \geq (\mathrm{n} - \max(\mathrm{m})) = (7 - 6)$ and in consequence, if a solution exists equation 3.7 will have infinitely many solutions.

However, the existance of solution to the actual system of equations, and not its associated homogeneous system, is not guaranteed:

Theorem 7. *For a linear system of equations $Ax = b$, a solution exists for all b if and only if the rank of A is equal to the dimension of b, i.e, A is full rank.*

This means that every time $\mathbf{J}(\mathbf{q})$ loses a rank, $r < m$, equation 3.7 won't have a solution for all $\dot{x}_d$ and the ability to control one axis either of translation or rotation will lost. The positions q in which this situation is given are called singularities.

In table 3.4, we can see that all the possible situations for equation 3.7 are summarized by the last column. If a solution exists there are infinitely many solutions, with dimension $n - m$.

Table 3.4: Number of solutions of a linear system

		number of solution of the homogeneous system?	
		one	**infinitely many**
a particular solution exists?	**yes**	unique solution	infinitely many solutions
	no	no solutions	no solutions

Summing up, the kinematic control problem has been defined as a set of linear equations (equation 3.7) and the existance of the solutions has been studied until now. Next, we must find a solution to the set of equations.

The most widely tool used to find the solution is the **Moore-Penrose pseudoinverse** of the jacobian $\mathbf{J}^{+}(\mathbf{q})$, where $\mathbf{J}^{+}(\mathbf{q}) = \mathbf{J}^{\top}\left(\mathbf{q}\right)(\mathbf{J}(\mathbf{q})\mathbf{J}^{\top}(\mathbf{q}))^{-1}$. The complete set of solutions can be described as:

$$\dot{\mathbf{q}} = \mathbf{J}^{+}(\mathbf{q})\dot{\mathbf{x}}_{\mathbf{d}} + \left(\mathbf{I} - \mathbf{J}^{+}(\mathbf{q})\mathbf{J}(\mathbf{q})\right)\boldsymbol{\xi} \tag{3.8}$$

$\boldsymbol{\xi} \in \mathbb{R}^{n\times 1}$ is the free variable that lets us choose between different $\dot{\mathbf{q}}$ to obtain the same end effector velocity $\dot{\mathbf{x}}_{\mathbf{d}}$. This is possible because $(\mathbf{I} - \mathbf{J}^{+}(\mathbf{q})\mathbf{J}(\mathbf{q}))$ is the null space projection matrix of $\mathbf{J}(\mathbf{q})$, meaning that it projects any vector into its null space. This implies that it doesn't change the final desired velocity.

This solution give us all the posible solutions for the least squares problem, which aims to minimize $\|\dot{\mathbf{x}}_{\mathbf{d}} - \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}\|$. More specifically, if $\boldsymbol{\xi} = \mathbf{0}$, $\dot{\mathbf{q}} = \mathbf{J}^{+}(\mathbf{q})\dot{\mathbf{x}}_{\mathbf{d}}$ is the minimum norm solution to the least squares problem.

However, there are some problems with this solution. As mentioned before, $\mathbf{J}(\mathbf{q})$ not being full rank implies the control over one of the motion axes is lost. This happens at certain joint

configurations $\mathbf{q}$ called singularities. Not only is reaching one of these configurations a problem, but near to singularity configurations are also problematic. This is true because even though a solution is possible, at these near to singularity configurations excessive joint space velocities are required to move the end effector in the axis that is close from being uncontrolable.

The reader may think that it would be useful to have a measure of when one of how close the configuration of the robot is from a singularity. One possible solution is the one introduced by [27]:

$$\sqrt{\det(\mathbf{J}^{\intercal} \cdot \mathbf{J})}$$

This measure indicates how close a robot is from a singularity. It is the equivalent to the determinant of a square matrix. When it approaches 0 it indicates $\mathbf{J}$ is close from loosing a rank.

Some method needs to be used to avoid reaching a singularity configuration. Recall that there are infinitely many solutions to the joint velocities that accomplish the desired end effector velocity $\dot{\mathbf{x}}_{\mathbf{d}}$, so if the addecuate joint velocities are chosen singularities can be avoided.

[28] presented a method that makes use of the Lagrange multipliers to minimize a certain objective function $h(\boldsymbol{q})$. This method says that introducing the following expression in $\boldsymbol{\xi}$ in equation 3.8 will decrease the value of the chosen objective function:

$$\boldsymbol{\xi} = -\alpha \nabla_{\mathbf{q}} h(\boldsymbol{q})$$

where α is a weight that allows to increase or decrease the effect of this second term on the final solution. Small values of α will be unnoticeable, while if the values are too big the value of $h(\boldsymbol{q})$ could even increase, since the gradient is only valid in the surroundings of the configuration at a certain time.

One possible objective function that can avoid singularities is:

$$h(\boldsymbol{q}) = \frac{1}{2} \sum_{i=1}^{n} \left(\frac{q_i - q_{i,\text{mid}}}{q_{i,\max} - q_{i,\min}} \right)^2$$

This function approaches 0 as the joints are close to their mid range value. The reason why this is a valid approach is that singularities usually appear when joints are close their limits.

3.3.1.6 End effector Position Control through Inverse Instantaneous Kinematics

In this section we present a simple algorithm that moves the end effector to a desired position in the cartesian space, shown in the function `moveToPosition` in listing 3.12. The mathematical elements presented in previous sections are an essential part of the code.

```
void moveToPosition(const Eigen::Vector3d xd)
{
    Eigen::VectorXd qDot;
    positionErrorVector = xd - x;

    while (positionErrorVector.norm() > position_error_threshold
            && ros::ok())
    {
        readEndEffectorPosition();
        readControlPointPositions();
        positionErrorVector = xd - x;
        xdDot = pGain * positionErrorVector;
        ros::spinOnce();
        qDot = IIK(xdDot)
        jointVelocityController.sendVelocities(qDot);
        rate.sleep();
    }
    qDot = Eigen::VectorXd::Constant(7, 0.0);
    jointVelocityController.sendVelocities(qDot);
}

Eigen::VectorXd IIK(Eigen::Vector3d xdDot)
{
    J = kdlSolver.computeJacobian("end_effector", q);
    Jpinv = J.completeOrthogonalDecomposition().pseudoInverse();
    Eigen::VectorXd qDot1, qDot2;
    qDot1 = Jpinv * xdDot;
    qDot2 = secondaryTaskGain *
            ((Eigen::MatrixXd::Identity(7,7) - Jpinv*J) *
            secondaryTaskFunctionGradient(q));
    return qDot1 + qDot2;
}

Eigen::VectorXd secondaryTaskFunctionGradient(Eigen::VectorXd q)
{
    Eigen::VectorXd qMid, qRanges;
```

```
    qMid = 0.5 * (jointLimitsMax + jointLimitsMin);                    37
    qRanges = jointLimitsMax - jointLimitsMin;                         38
    return 2.0/7.0 * (q - qMid).cwiseQuotient(qRanges);                39
}                                                                      40
```

Listing 3.12: CartesianPositionController.cpp

Note that the velocity we command at every control loop is $()\mathbf{x_d} - \mathbf{x})$, that is a vector that points from the current end effector possition to the desired objective point. This means that the trajectory of the end effector will be a straight line that goes from the initial point to the objective point.

This simple method will be used as the base desired velocity for all the avoidance control methods.

3.3.2 Collision avoidance algorithm based on potential field methods and joint velocity constraints

The method presented in this section is an adaptation of the academic article "A Depth Space Approach to Human-Robot Collision Avoidance" [29], published by F. Flacco et al. in 2012. The paper presents an avoidance algorithm that takes depth images as inputs. In this work we have implemented the same functionality leveraging the perception via proximity sensing presented in section 3.2 instead. Moreover, the resolution of the equations is done via optimization, whereas in the original work it was done using a more complex algorithmic approach that leveraged inverse instantaneous kinematics explained in 3.3.1.5.

The main task of robot control usually deals with the motion of the end effector. The objective of the collision avoidance control algorithm is to maintain the original main task as long as collision can be avoided.

In consequence, a disctintion is made between a discrete set of points in the body of the robot and the end effector. The points in the **body** are taken to avoid the collision of any part of the robot with an object in its surroundings. On the one hand, the **end effector** is given special attention due to its aforementioned importance.

The end effector algorithm is based on a **potential field** approach, where artificially generated repulsive fields avoid getting closer to a possible collision. This algorithm is explained in detail in section 3.3.2.4.

The body control points are treated as cartesian constraints that are translated into **joint velocity constraints**. This constraints can be satisfied thanks to task redundancy, that is, there are different joint velocities that move the end effector at the same velocity. The authors state that if they had considered the same approach as for the end effector, they would have had to prioritize some points over the rest so that the problem turned feasible. However, doing this is problematic in their own words: giving priority to the end effector the collision safety of the points in the body would not be guaranteed, whereas not prioritizing it would result in random end effector trajectories that would not accomplish the task. The obtainment of these joint velocity constraints is addressed in section 3.3.2.2.

3.3.2.1 Potential field method approach applied to the effector obstacle avoidance

Let $\dot{\mathbf{x}}_\mathbf{d}$ be the desired velocity for the end effector. This velocity can be coming from different sources, but one example is the velocity that points to the next point in a cartesian position control3.3.1.6. The location of the end effector is described by the position vector relative to the base frame $\mathbf{x_{ee}}$. In addition, the robot is surrounded by a set of m obstacles, represented by their position vectors relative to the base frame of the robot $\mathbf{h_i}$, for $i = 1, 2, \ldots, m$.

The algorithm presented in this section will adapt this velocity to be reactive to obstacles, thanks to the generation of an artificial repulsive vector $\dot{\mathbf{x}}_\mathbf{rep}$. This repulsive vector will be used to generate the final control velocity $\dot{\mathbf{x}}_\mathbf{c}$:

$$\dot{\mathbf{x}}_\mathbf{c} = \dot{\mathbf{x}}_\mathbf{d} - \dot{\mathbf{x}}_\mathbf{rep}$$

In order to obtain $\dot{\mathbf{x}}_\mathbf{rep}$, firstly the distance vectors for all the obstacles are calculated:

$$\mathbf{d_i} = \mathbf{h_i} - \mathbf{x_{ee}}$$

Among all these vectors, the smallest one is selected. This vector is denoted with $\mathbf{d_{min}}$ and it is the only factor that affects direction and norm of $\dot{\mathbf{x}}_{\mathbf{rep}}$.

$$\dot{\mathbf{x}}_{\mathbf{rep}} = V \cdot \frac{1}{1 + e^{\alpha(2\frac{\|\mathbf{d_{min}}\|}{D} - 1)}} \cdot \frac{\mathbf{d_{min}}}{\|\mathbf{d_{min}}\|} \tag{3.9}$$

The direction of the vector is exactly the same as that of $\mathbf{d_{min}}$. The parameters that affect the shape of the curve that describes the norm of the repulsive vector are V, D and α.

- V is the maximum value the norm of the repulsive vector can take, since $\frac{1}{1+e^{\alpha(2\frac{\|\mathbf{d_{min}}\|}{D}-1)}}$ is bounded between 0 and 1.
- D is the minimum distance at which the norm of the repulsive vectors starts increasing. For distances greater than D the norm is 0.
- α defines the form of the curve. As α tends to ∞, the shape of the curve tends to that of of a step function.

This interactive graph allows fine tuning the parameters.

3.3.2.2 Joint velocity constraints to avoid collision of any point in the robot

The robot body must also avoid obstacles. For that purpose, each of the control points that describe the robot body will provide a set of joint velocity limits.

Let one of those control points be described by its position vector with respect to the base frame $\mathbf{c_j}$. The set of obstacles is the same as in the previous section.

The procedure to obtain the joint velocity limits for a given control point $\mathbf{c_j}$ starts in the same way as in the end effector, that is, finding the closest obstacle to $\mathbf{c_j}$ and obtaining the correspoinding distance vector $\mathbf{d_{min}}$.

Then f is calculated. This expression is identical to part of equation 3.9.

$$f = \frac{1}{1 + e^{\alpha(2\frac{\|\mathbf{d_{min}}\|}{D} - 1)}}$$

This value is bounded between 0 and 1 and is close to 1 when the probability of hitting an object is high. The shape of the curve that defines f is exactly the same as that of the curve from the previous section.

Then, vector $\mathbf{s}$ is calculated:

$$\mathbf{s} = \mathbf{J_{c_i}}^{+} \frac{\mathbf{d_{min}}}{\|\mathbf{d_{min}}\|} f$$

$\mathbf{J_{c_i}}$ is the partial jacobian of the kinematic chain that goes from the base to $\mathbf{c_j}$. Not all the joints necesarily influence the movement of that control point. For instance, if the control point is located in link 2 (see Denavit-Hartenberg notation in 3.3.1.1 for more information on link numbering) only joints 1 and 2 affect the movement of that point. Therefore, the size of $\mathbf{J_{c_i}}$ and, in consequence, of $\mathbf{s}$ depend on the location of the specific control point we are talking about.

The ith value of $\mathbf{s} \in \mathbb{R}^{n \times 1}$ represents the influence the movement of that specific joint i has in moving $\mathbf{c_j}$ as indicated by the vector $\frac{\mathbf{d_{min}}}{\|\mathbf{d_{min}}\|}$. The sign of s_i represents the direction of rotation of that joint that leads to that movement. If $s_i > 0$, moving that joint counterclockwise would contribute to a collision. On the other hand, if $s_i < 0$ hte clockwise rotation of the joint is the one that has to be limited.

The amount by which joint velocities are limited depends on the value of f and the physical limits of the robot (joint velocity limits impose that $-\dot{\mathbf{q}}_{\max} \leq \dot{\mathbf{q}} \leq \dot{\mathbf{q}}_{\max}$, see 3.1.1):

$$\begin{aligned} &\text{if } s_i \geq 0, \quad \dot{q}^{j}_{\max,body,i} = \dot{q}_{\max,i}\,(1-f) \\ &\text{else} \qquad\quad \dot{q}^{j}_{\min,body,i} = -\dot{q}_{\max,i}\,(1-f) \end{aligned}$$

Once $\dot{\mathbf{q}}^{\mathbf{j}}_{\min,\mathbf{body}}$ and $\dot{\mathbf{q}}^{\mathbf{j}}_{\max,\mathbf{body}}$ have been obtained for all control points, the final restrictions $\dot{\mathbf{q}}_{\min,\mathbf{body}}$ and $\dot{\mathbf{q}}_{\max,\mathbf{body}}$ need to be set as the the most restrictive.

$$\begin{aligned} \dot{q}_{\min,body,i} &= \max\{\dot{q}^{j}_{\min,body,i}\} \\ \dot{q}_{\max,body,i} &= \min\{\dot{q}^{j}_{\max,body,i}\} \end{aligned}$$

3.3.2.3 Imposing joint position, velocity and acceleration limits

The aforementioned approach doesn't take into account the physical limitations of the robot. Positions of the joints are limited, as mentioned in section 3.1.1. In addition, repulsive vectors can sometimes provoke unpredictable sudden changes in the motion of the robot. This can be avoided limiting the accelerations of the joints. The previous section has explained how joint velocity limitations can be used to avoid obstacles.

To limit joint positions, and accelerations, we assume that the restrictions that these can be written as follows:

$$\mathbf{q}_{\min} \leq \mathbf{q} \leq \mathbf{q}_{\max}$$

$$-\ddot{\mathbf{q}}_{\max} \leq \ddot{\mathbf{q}} \leq \ddot{\mathbf{q}}_{\max}$$

Position and acceleration restrictions can be brought to the velocity level $\dot{\mathbf{q}}$. This is a necessary step since, as presented in the next section, the whole problem will be defined as a joint velocity optimization problem. In [30], they present a method to express these limits in the velocity level:

$$\begin{array}{ll} \textbf{Position limits} & \frac{q_{\min,i}-q_i}{T} \leq \dot{q}_i \leq \frac{q_{\max,i}-q_i}{T} \\ \textbf{Acceleration limits} & -\sqrt{2\ddot{q}_{\max,i}\left(q_i - q_{\min,i}\right)} \leq \dot{q}_i \leq \sqrt{2\ddot{q}_{\max,i}\left(q_{\max,i} - q_i\right)} \end{array} \tag{3.10}$$

where q_i is the current position of joint i and the is T is the control period.

Finally, the most restrictive values between position, acceleration and the body algorithm restrictions need to be selected.

$$\begin{aligned} \dot{q}_{\min,i} &= \max\left\{\tfrac{q_{\min,i}-q_i}{T}, \dot{q}_{\min,body,i}, -\sqrt{2\ddot{q}_{\max,i}\left(q_i - q_{\min,i}\right)}\right\} \\ \dot{q}_{\max,i} &= \min\left\{\tfrac{q_{\max,i}-q_i}{T}, \dot{q}_{\max,body,i}, \sqrt{2\ddot{q}_{\max,i}\left(q_{\max,i} - q_i\right)}\right\} \end{aligned} \tag{3.11}$$

3.3.2.4 Setting up the optimization problem

Although this paper was not originally implemented using optimization, we believe that this is an easier and more intuitive way of describing the desired end effector velocity and the joint

velocity descriptions.

Therefore, the purpose of this section is to define the optimization problem that takes the modified end effector velocity $\dot{\mathbf{x}}_\mathbf{c}$ obtained in section 3.3.2.4 and the restrictions obtained in section 3.3.2.2 as inputs and outputs the joint velocities $\dot{\mathbf{q}}$.

Firstly, we need to describe the objective function in 3.1. The optimal $\dot{\mathbf{q}}^*$ that minimizes the objective funtion must be one that moves the end effector with the desired velocity $\dot{\mathbf{x}}_\mathbf{c}$. This is the proposed form for the objective function (from now on we will denote $\dot{\mathbf{x}}_\mathbf{c}$ with simply $\dot{\mathbf{x}}$):

$$f(\dot{\mathbf{q}}) = \underbrace{\frac{1}{2}(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}})^\mathbf{T}(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}})}_{\mathbf{1}} + \underbrace{\frac{\mu}{2}\dot{\mathbf{q}}^\mathbf{T}\dot{\mathbf{q}}}_{\mathbf{2}}$$

.

(1) This is the quadratic error between the robot's target velocity and its actual motion given the optimization variables $\dot{\mathbf{q}}$.

$$\frac{1}{2}\left(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}\right)^\mathrm{T}\left(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}\right) = \frac{1}{2}\|\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}\|^2$$

Observe above that $\dot{\mathbf{x}}$ represents the desired motion, while $\mathbf{J}\dot{\mathbf{q}}$ represents the actual motion determined by the jacobian $\mathbf{J}$ presented in section 3.3.1.4. Therefore, $\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}$ is the difference between the desired and actual motions. Performing the actual matrix math on the first term gives us the scalar on the right in the equation above, which happens to be the square of the norm of the error, hence the quadratic error.

Note that the solution to the optimization problem $\dot{\mathbf{q}}$ would be the result of doing $\dot{\mathbf{q}} = \mathbf{J}^\dagger\dot{\mathbf{x}}$, as explained in section 3.3.1.5 if no further constraints where applied to the optimization problem. That is, $\dot{\mathbf{q}} = \mathbf{J}^\dagger\dot{\mathbf{x}}$ would be the smallest value vector to solve $\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}$ and minimize the term $||\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}||$.

(2) As mentioned in section 3.3.1.5, problems appear with the jacobian when near singularities. This second term is used to solve this issue using the **Damped Least Squares** method

[31] [32].

The value of μ is calculated as follows, based in the manipulability measure (section 3.3.1.5) :

$$w = \sqrt{\det\left(\mathbf{J}\mathbf{J}^\top\right)}$$
$$\mu = \begin{cases} \mu_0 \left(1 - \frac{w}{w_0}\right)^2, & w < w_0 \\ 0, & w \geq w_0 \end{cases}$$

This expression needs to be expanded to meet the the quadratic form of the objective function in 3.1.

$$\begin{aligned} f(\dot{\mathbf{q}}) &= \frac{1}{2}(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}})^\top(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}) + \frac{\mu}{2}\dot{\mathbf{q}}^\top\dot{\mathbf{q}} \\ &= \frac{1}{2}\underbrace{(\dot{\mathbf{x}}^\top - (\mathbf{J}\dot{\mathbf{q}})^\top)}_{(\mathbf{A}+\mathbf{B})^\top = \mathbf{A}^\top + \mathbf{B}^\top}(\dot{\mathbf{x}} - \mathbf{J}\dot{\mathbf{q}}) + \frac{\mu}{2}\dot{\mathbf{q}}^\top\dot{\mathbf{q}} \\ &= \frac{1}{2}[\dot{\mathbf{x}}^\top\dot{\mathbf{x}} - \underbrace{\dot{\mathbf{x}}^\top\mathbf{J}\dot{\mathbf{q}} - (\mathbf{J}\dot{\mathbf{q}})^\top\dot{\mathbf{x}}}_{\substack{\textbf{Both equivalent to:} \\ \dot{\mathbf{x}}\cdot\mathbf{J}\dot{\mathbf{q}} \\ \textbf{since for scalar value } a \\ a^\top = a}} + (\mathbf{J}\dot{\mathbf{q}})^\top(\mathbf{J}\dot{\mathbf{q}})] + \frac{\mu}{2}\dot{\mathbf{q}}^\top\dot{\mathbf{q}} \\ &= \frac{1}{2}(\dot{\mathbf{x}}^\top\dot{\mathbf{x}} - 2\dot{\mathbf{x}}^\top\mathbf{J}\dot{\mathbf{q}} + \underbrace{\dot{\mathbf{q}}^\top\mathbf{J}^\top}_{\substack{(\mathbf{A}\mathbf{B})^\top \\ = \mathbf{B}^\top\mathbf{A}^\top}}\mathbf{J}\dot{\mathbf{q}}) + \frac{1}{2}\dot{\mathbf{q}}^\top\mu\mathbf{I}\dot{\mathbf{q}} \\ &= \frac{1}{2}\dot{\mathbf{x}}^\top\dot{\mathbf{x}} - \dot{\mathbf{x}}^\top\mathbf{J}\dot{\mathbf{q}} + \frac{1}{2}\dot{\mathbf{q}}^\top(\mathbf{J}^\top\mathbf{J} + \mu\mathbf{I})\dot{\mathbf{q}} \\ &= \frac{1}{2}\dot{\mathbf{q}}^\top\left(\mathbf{J}^\top\mathbf{J} + \mu\mathbf{I}\right)\dot{\mathbf{q}} - \dot{\mathbf{x}}^\top\mathbf{J}\dot{\mathbf{q}} + \frac{1}{2}\dot{\mathbf{x}}^\top\dot{\mathbf{x}} \end{aligned}$$

$\frac{1}{2}\dot{\mathbf{x}}^\top\dot{\mathbf{x}}$ is a constant w.r.t. the optimization variables in $\dot{\mathbf{q}}$, so even though it will change the inal value of the objective function, it does not affect $\dot{\mathbf{q}}^*$. Since the resulting $\dot{\mathbf{q}}^*$ is the only important output of the optimization process regarding our application, we can get rid of this term, resulting in the final expression:

$$f(\dot{\mathbf{q}}) = \frac{1}{2}\dot{\mathbf{q}}^\top\left(\mathbf{J}^\top\mathbf{J} + \mu\mathbf{I}\right)\dot{\mathbf{q}} - \dot{\mathbf{x}}^\top\mathbf{J}\dot{\mathbf{q}}$$

Comparing this expanded expression to 3.1, we have that:

$$\mathbf{Q} = \mathbf{J}^\top \mathbf{J} + \mu \mathbf{I}$$
$$\mathbf{c}^\top = -\dot{\mathbf{x}}^\top \mathbf{J}$$

As stated in section 3.1.4, the problem needs to be convex in order to be able to find a global optimal solution in a minimum amount of time. This will only be the case if we can verify $\mathbf{Q}$ will always be semi-positive definite, or what is the same, its eigenvalues will be non-negative.

Theorem 8. *Let $\mathbf{A} \in \mathbb{R}^{m \times n}$. Then, $\mathbf{A}\mathbf{A}^\top$ is a positive semi-definite matrix.*

Theorem 9. *The sum of any two (semi)positive definite matrices of the same size is (semi)positive definite.*

$\mu\mathbf{I}$ is semi-positive definite because it is a diagonal matrix with non-negative coefficients and the eigenvalues of a diagonal matrices are the coefficients themselves. $\mathbf{J}^\top \mathbf{J}$ because of theorem 8. Then, by theorem 9 we know that $\mathbf{Q}$ is semi-positive definite.

On the other hand, the restrictions presented in section 3.3.2.2 could be rewritten as 4 linear equations to meet the structure in equation 3.1. However, Quadratic Programming optimization algorithms can deal more efficiently with a different kind of constraint for this specific kind of restrictions, where only upper and lower boundaries for the optimization variables are required. This type of constraints are called **box constraints** and can be expressed as:

$$\dot{\mathbf{q}}_{\text{min}} \leq \dot{\mathbf{q}} \leq \dot{\mathbf{q}}_{\text{max}}$$

where the components of $\dot{\mathbf{q}}_{\text{min}}$ and $\dot{\mathbf{q}}_{\text{max}}$ are calculated with the expression 3.10.

Once the end effector velocity is set up in the objective funtion and the restrictions are applied, the resolution of the Quadratic Programming problem presented in 3.1 returns the best joint velocities that solve all the specifications presented in this section.

3.3.3 Alternative approach to collision avoidance based on linear inequalities

In this section we implement the academic article "Collision Avoidance with Proximity Servoing for Redundant Serial Robot Manipulators" [33], published by Y. Ding and U. Thomas in May 2020. In their approach box constraints are reserved to the task of maintaining the limits in joint positions, velocities and accelerations. The hierarchy between the algorithms applied to the end effector and the rest of the body is not used any more. Control points are calculated dynamicaly (section 3.3.3.1). The commanded velocity is not modified by a repulsive vector. Instead, each obstacle point defines a restriction on the movement of the closest point in the kinematic chain (section 3.3.3.2). An extra restriction makes the overall distance to the obstacles increase over time (section 3.3.3.3).

3.3.3.1 Assigning a control point in the kinematic chain of the robot to each of the obstacles

In this section control points are not predefined as in section 3.3.2. All the obstacle points are dynamicaly assigned a point in the robot's kinematic chain instead. The point is chosen to be the one closest to the obstacle.

The method implemented to obtain the closest point is presented in [34]. These are the steps proposed:

(1) Let the robot be represented by a series of segments defined by pairs of points. Let P be the obstacle point to which we want to assign the closest point in the robot.

(2) For each segment that goes from A to B:

   (a) Write the vector between P and a point that is t from A to B. (When $t = 0$, the point is A; when $t = 1$, the point is B.)

$$f(t) = (1 - t)A + tB - P \tag{3.12}$$

(b) Now let $g(t) = \|f(t)\|^2$, which is norm of that vector squared. We want to find the value of t that minimizes $g(t)$.

(c) Develop expression 3.12,

$$\begin{aligned} g(t) &= \|f(t)\|^2 \\ &= ((1-t)A + tB - P)^T \cdot (1-t)A + tB - P \\ &= (t(B-A) + A - P)^T \cdot (t(B-A) + (A-P)) \end{aligned}$$

Let $v = B - A$ and $u = A - P$,

$$\begin{aligned} g(t) &= \|f(t)\|^2 \\ &= (tv + u)^T \cdot (tv + u) \\ &= t^2\|v\|^2 + 2t(v \cdot u) + \|u\|^2 \end{aligned}$$

(d) Calculate $g'(t)$,

$$g'(t) = 2t\|v\|^2 + 2(v \cdot u) = 2t(v \cdot v) + 2(v \cdot u). \tag{3.13}$$

(e) Set to zero and solve to get the value of t that minimizes $g(t)$. This particular function has only one minimum, so if you find a critical point you know it is a minimum value.

$$t = -\frac{v \cdot u}{v \cdot v}.$$

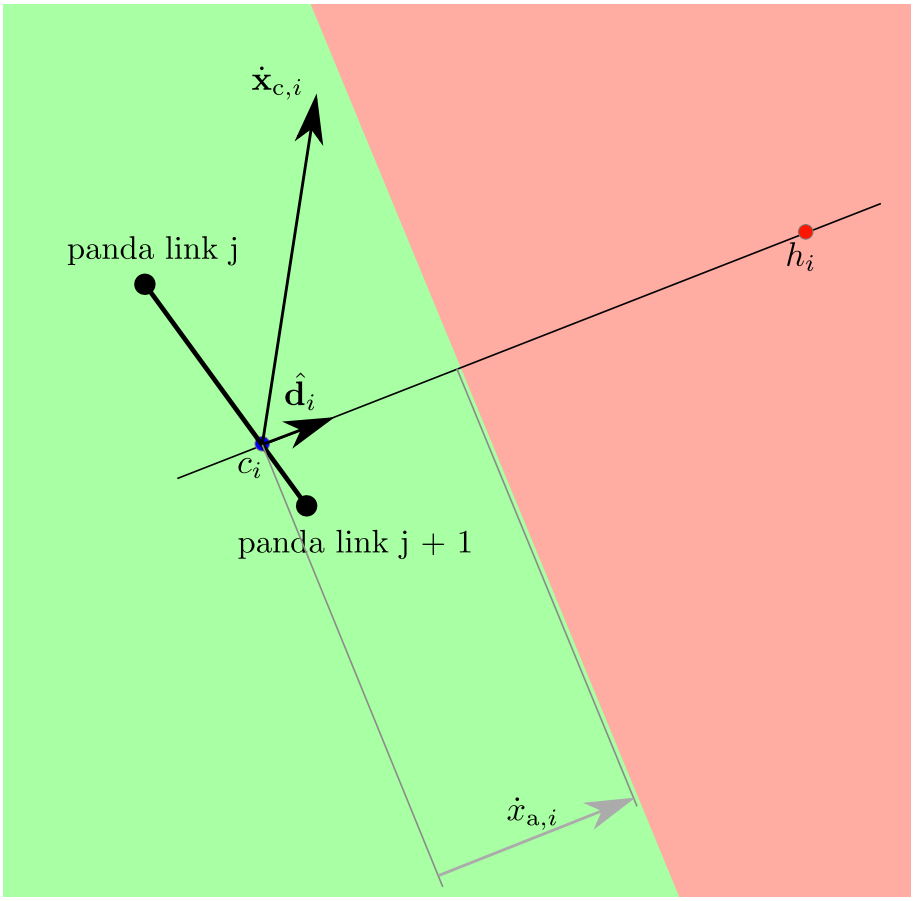
(f) Once the value of t is computed, if it is between 0 and 1, then the closest point is $(1-t)A + tB$. If it is not, A or B are the closest points, depending on which $f(0)$ or $f(1)$ is the smaller.

(3) From the list of points associated to each of the segments, choose the one corresponding to the minimum distance.

3.3.3.2 Limiting the approach velocity to the obstacles

The expression described in this section will restrict the approach velocity of the point assigned in section 3.3.3.1 towards the obstacle. The restriction is described graphically in figure 3.13.

Figure 3.13: Restriction of the approach velocity of a point towards the obstacle.

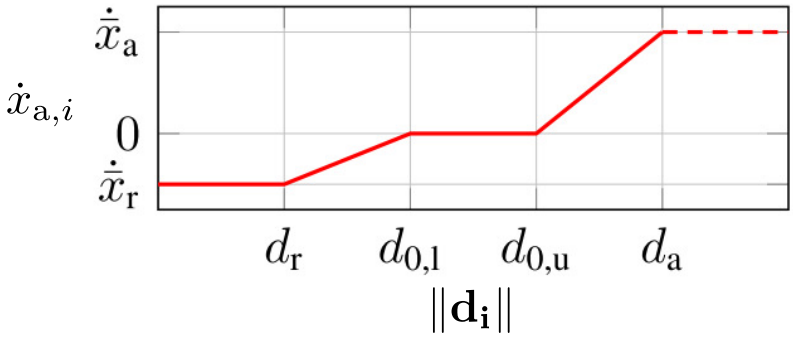


Given a position of an obstacle h_i, the orthogonal projection of the final velocity $\dot{\mathbf{x}}_{\mathrm{c},i}$ will be limited by a certain amount $\dot{x}_{\mathrm{a},i}$. Every $\dot{\mathbf{x}}_{\mathrm{c},i}$ whose arrowhead is within the green area will be permitted.

Moreover, if the value of $\dot{x}_{\mathrm{a},i}$ is negative, then the red boundary between the green and red areas will be moved to the opposite site of $\hat{\mathbf{d}}_i$, forcing a repulsive action.

Consequently, the value of $\dot{x}_{\mathrm{a},i}$ defines how the behaviour of a given point is restricted. In [33] they propose a piecewise defined function to calculate the value, based on the distance at which the obstacle is located with respect to the robotic arm, shown in figure 3.14.

Figure 3.14: Determination of $\dot{x}_{\mathrm{a},i}$ as a function of $\|\mathbf{d_i}\|$.



Formally speaking, the restriction for one obstacle is formulated as follows:

$$\hat{\mathbf{d}}_i^{\top} \dot{\mathbf{x}}_{\mathrm{c},i} = \hat{\mathbf{d}}^{\top} \mathbf{J}_{\mathrm{p_{c},}i} \dot{\mathbf{q}} \leq \dot{x}_{\mathrm{a},i}$$

This defines one linear inequation, where $\hat{\mathbf{d}}^{\top}\mathbf{J}_{\mathrm{p_{c},}i} \in \mathbb{R}^{1\times n}$ is the coefficients vector, $\dot{\mathbf{q}} \in \mathbb{R}^{n\times 1}$ are the unknowns and $\dot{x}_{\mathrm{a},i} \in \mathbb{R}$ is the right side of the inequation. As one obstacle defines one new restriction, for m obstacle points, m rows will be added to elements $\mathbf{A}$ and $\mathbf{b}$ in the Quadratic Programming problem definition in 3.1.

It has to be noted that the restrictions for several points can be contradictory sometimes. This turns the optimization unfeasible. When this happens, the repulsive actions are set to 0 and the optimization is run again.

3.3.3.3 Increasing the overall distance to the obstacles over time

In this section a new restriction is described, which forces an overall increase over time of the distance between the obstacles and the robot. This restriction complements the one presented in section 3.3.3.2.

In terms of mathematical analysis, a continuous function is increasing in time when its derivative with respect to time is positive. By the chain rule, this is how the derivative of a single distance norm with respect to time would be calculated:

$$\frac{d\|\mathbf{d_i}\|}{dt} = \frac{\partial\|\mathbf{d_i}\|}{\partial\mathbf{q}}\frac{d\mathbf{q}}{dt} = \nabla_{\mathbf{q}}\|\mathbf{d_i}\|\dot{\mathbf{q}}$$

As what we want is an overall increase of the distance instead of increasing just one distance, a weight based formulation can be employed:

$$\begin{bmatrix} w_1 & \cdots & w_m \end{bmatrix} \begin{bmatrix} \frac{d\|\mathbf{d_1}\|}{dt} \\ \cdots \\ \frac{d\|\mathbf{d_m}\|}{dt} \end{bmatrix} = \begin{bmatrix} w_1 & \cdots & w_m \end{bmatrix} \begin{bmatrix} \nabla_{\mathbf{q}}\|\mathbf{d_1}\| \\ \cdots \\ \nabla_{\mathbf{q}}\|\mathbf{d_m}\| \end{bmatrix} \dot{\mathbf{q}} \geq 0$$

The weights are set to be indirectly proportional to the distance, so that closest obstacles have highest priority:

$$w_i = \frac{1}{\|\mathbf{d_i}\|}$$

This constraint adds one last row to $\mathbf{A}$ and $\mathbf{b}$.

Chapter 4

Results

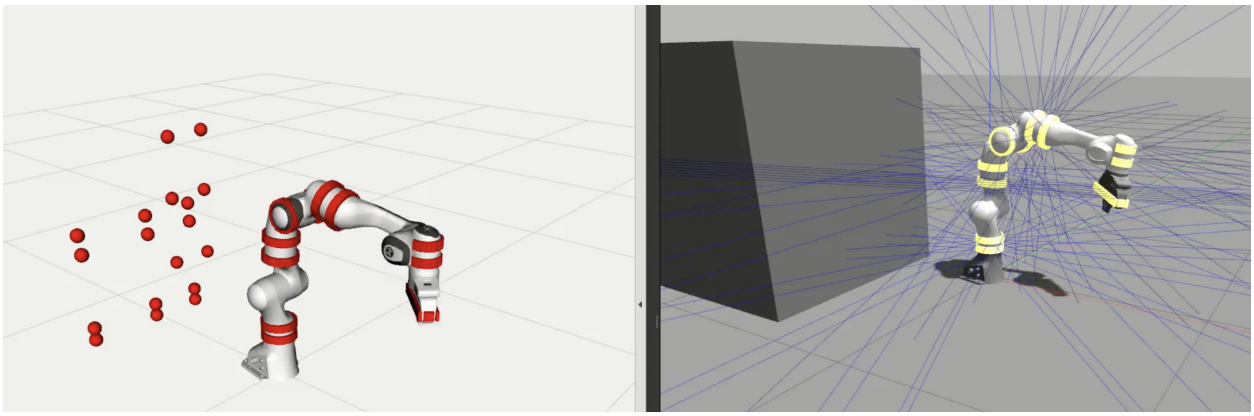
In this section we present the results of the implementation of all the elements presented in chapter 3.

4.1 Perception

The culmination of the proximity sensor transformation explained in section 3.2, in addition to the work done in calibration by the calibration group in the HIRO Lab, led by Kandai Watanabe, will allow to the robot to sense its surroundings.

In simulation, we show below that proximity sensors placed in the robot body (marked in red in figure 3.9) can detect objects in simulation. In figure 4.1, `rviz` is placed on the left and the simulation in `Gazebo` on the right. A black box is moved around in `Gazebo` and the robot's perception, result of the transformation algorithm proposed in this work, is seen in `rviz`.

Figure 4.1: Final visualization of the perceived points in *rviz*.



This video shows the same result in movement. Note that the perception is instantaneous. `Gazebo` updates the position of the box when the left button of the mouse is released and this is the precise time that the old red spheres are deleted from perception and the new ones are created.

The update rate achieved in simulation has been 100 Hz. In practical applications the rate is limited by the specific sensor's capabilites, as opposed to the fact that the limiting factor is the computer's processing time for depth sensors, sensors that traditionally have been use for the obstacle avoidance control.

The buffer size in the demonstration is 10. At 100 Hz, this buffer size means it holds the data for 100 ms.

4.2 End effector position control using the general solution of Instantaneous Forward Kinematics

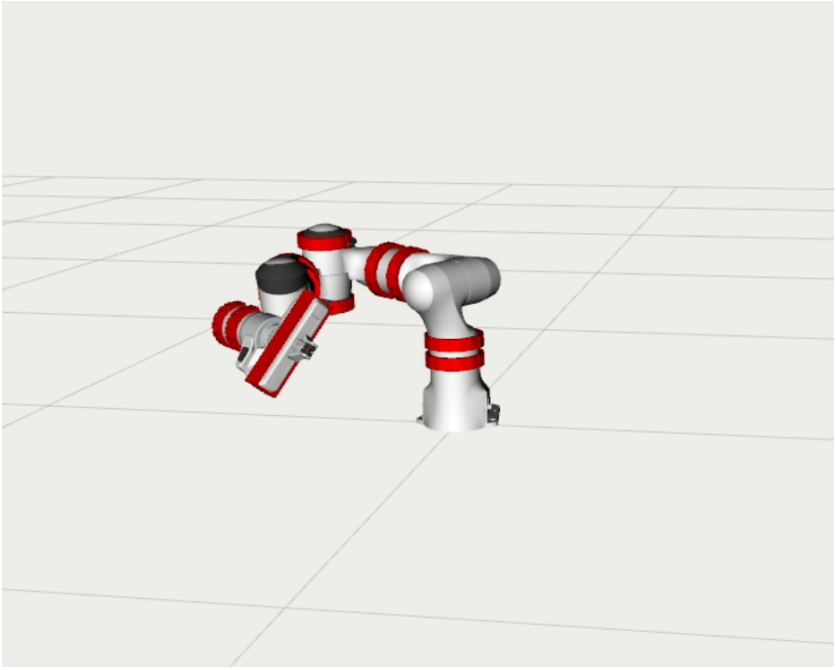
Cartesian position control, whose final code has been shown in section 3.3.1.6 was a necessary step to implement the rest of the avoidance algorithms. We have used many elements implemented in this controller as a foundation and to inform the movement of all subsequent controllers.

This video presents the control in action. The desired trajectory is composed of 4 points that pertain to a circunference in the $x = 0.5$ m plane. This video presents the experiment that will be

conducted for a first evaluation of the avoidance algorithm. The red point represents an obstacle and obviously it is not avoided in the case of this simple controller.

The gradient of the objective function that represents how close the joints are with respect to their mid range value, that goes into the second term of equation 3.8, is a key element in this controller. If this them was omitted, weird joint configurations can be reached. As shown in figure 4.2 and This video, starting from a good initial position everything seems to work well in the beginning. However, at the second half of the video the arm reaches an undesired configuration in which it is near to hit the ground. This happens because of all posible joint velocities the minimum is being selected and this does not always involve the best solution. For a more detailed explanation on how the gradient works see section 3.3.1.5.

Figure 4.2: Panda with the gradient method deactivated.



Another important result is that, in listing 3.12, if the threshold that decides if the main position control loop is complete is too small, the sensor update interval is not fast enough to ensure

that level of precision. The velocity commanded to the end effector also needs to be considered, since for higher velocities a greater amount of distance is covered in the same amount of time. For example, when this threshold is set to less than 1 cm for an end effector speed of 0.35 m/s , the final part of the trajectory sometimes becomes sloppy.

This threshold method is not the best one to solve the problem of trajectory control. A more advanced control method has been developed for the future development of the project. In contrast, it is a valid method for an initial checking of the behavior of the avoidance algorithms.

4.3 Collision avoidance algorithm based on potential field methods and joint velocity constraints

The approach explained in section 3.3.2 divided the avoidance control into two parts. One for the end effector and another one for the rest of the robot's body. We will demonstrate how this approach works in two parts. Firstly, the end effector algorithm will be tested independently. Then, the experiment presented in section 4.2

In this video we show an example of the repulsive vector modifying the trajectory of the end effector. The experiment is simple: the end effector is commanded an iterative movement between two points in a line parallel to the x axis. An obstacle is placed in the trajectory.

The result is that the repulsive vector modifies the trajectory in the beginning and successfully moves around the object. When moving torwards the second objective the end effector stops when the desired velocity and the repulsive vector are exactly the same but with opposite directions. This represents a local minimum of the problem, since the desired position is actually reachable while avoiding the obstacle.

In this other video, the repulsive method is combined with the joint restriction method and tested with the aforementioned experiment (following a trajectory between 4 points). The obstacle is now succesfully avoided. However, it can be observed how the joint restrictions sometimes limit movements that do not have to be limited.

Acceleration limits (section 3.3.2.3) have also been proven to be an important part of the

correct behavior of the control. In the process of implementation, fast changes in the repulsive vector's direction resulted in jerky movements. Limiting acceleration has been proven to be a suitable solution to this fact.

The methods demonstrated in the next sections will improve this limitation imposed by the restrictions that prevent collision.

4.4 Alternative approach to collision avoidance based on linear inequalities

The approach explained in section 3.3.3 different in two main ways with respect to Flacco. Control points where dynamically calculated for every obstacle point. The original desired end effector velocity was maintained as long as possible, as opposed to using repulsive vectors.

In this video the behaviour of the controller is shown. The first time the robot arm approaches the obstacle the avoidance is satisfactory. The second time something different happens. The movement torwards the next trajectory point is not allowed by the restrictions. The reason why this takes place is obvious if we analyze the second restriction (section 3.3.3.3). As we explained there, an overall increase in the distance to the obstacles is pursued by that restriction. In this case, as there is one single obstacle, it is the distance to that obstacle that is forced to increase. That is why the robot arm is not allowed to advance in the trajectory, there is no way to do so while satisfying the restriction.

On the other hand, another anomaly can be observed in the movement of the robot. After avoiding the obstacle for the first time, the position of the arm stays in the position adopted in order to avoid the obstacle. This effect was also observable in the previous control approach. What makes the robot approach the ground in figure 4.2 is the reason why this happens. The general solution shown in equation 3.8 and the gradient that minimizes the distance from the joint angles to their mid ranges are not being used in this section. In fact, the optimization selects the minimum norm joint velocity. This is exactly what happened if the second term of the general solution was deleted.

In addition, acceleration limitation is also very important in this section. The behaviour

thanks to these limitations is more predictable and safer.

4.5 Improvements to the collision avoidance control

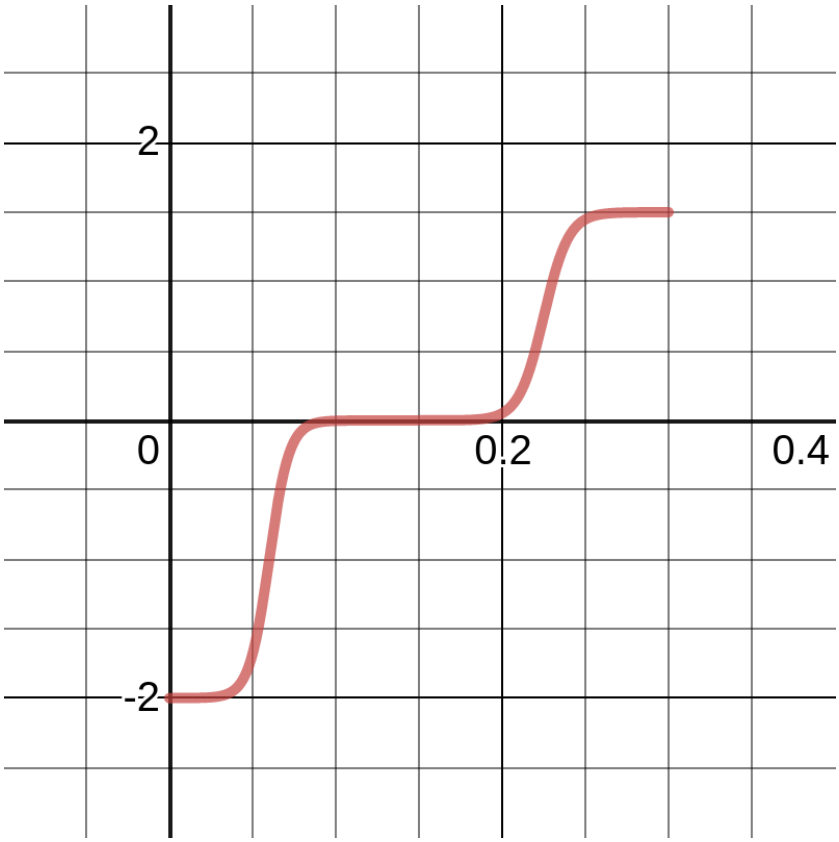
The restriction that causes the undesired behaviour in the previous test can be deleted. We have shown its implications are not convenient. Furthermore, we think that the restriction of the approach velocity is enough for if exploited properly. In section 3.3.3.2, we explained why the form of these restrictions are a suitable way of describing the avoidance problem. They allow setting up a repulsive action for any point in the robot when needed. This implies that the implications of the repulsive vectors applied to the end effector in section 3.3.2 can be extended to the rest of the body.

The original problem of maintaining the task as long as it is possible is also better described by this formulation, since the commanded velocity will be the original task velocity. That is, no repulsive vectors will modify the original task velocity. Since this goes in the optimization function, when it is not possible to maintain the original velocity, the closest possible velocity will be commanded.

In conclusion, the control proposed in this work is similar to the one presented in section 3.3.3, but it is a more fundamental formulation of the original description of the problem.

The graph used to model the repulsive actions in the original publication (figure 3.14) has been enhanced for a more natural behaviour (figure 4.3). See this interactive graph for a deeper understanding of the parameters. Instead of using a linear graph we propose using exponential functions. In addition we propose that there are two thresholds for making decisions: the distance at which the control starts taking into account the obstacle: $d_{noticeable}$ and the distance at which repulsive actions start to be considered $d_{critical}$.

Figure 4.3: Enhanced determination of $\dot{x}_{a,i}$ as a function of $\|\mathbf{d_i}\|$.



The result of this enhanced approach can be observed in this video.

4.6 Collision avoidance control in the real robot

All the implementations and experiments described in the document so far have been done in simulation. The final will be to verify all the control algorithms work as well in the real Franka Panda Emika.

This video shows the same experiment carried on in the last section, but in real life.

The perception algorithm explained in 3.2 has not been tested. The development of the artificial skin has been parallel to this work and as of the day of completion, the electronics of the sensors are in their third version and waiting to be tested. The kinematic calibration method, in contrast, has been succesfully tested using individual *IMU* sensors.

The obstacle in the video has been placed manually via de command line tool `rostopic pub`. We show the location of the point with our hand.

Chapter 5

Conclusions

one short paragraph about each section

future work: where we're going.

Bibliography

[1] iRobot. Roomba vacuuming robot, 2020. https://www.irobot.com/.

[2] thermomix. thermomix cooking robot, 2020. https://www.thermomix.com/tm6/?utm_source=google&utm_medium=cpc&utm_term=thermomix&utm_campaign=brand&utm_content=ETA&gclid=Cj0KCQjwjer4BRCZARIsABK4QeWxAdizt1-6N-cJk1-wBOjGrsYqi0lptmd7M8yAm1P-vG1YVJzZtyEaAkGjEALw_wcB.

[3] Diligent Robots. Moxi hospital robot assistant, 2020. https://diligentrobots.com/moxi.

[4] Avatarmind. ipal robot family, 2020. https://www.ipalrobot.com/.

[5] G Westling and RS Johansson. Factors influencing the force control during precision grip. Experimental brain research, 53(2):277–284, 1984.

[6] Jan BF van Erp and Hendrik AHC van Veen. Touch down: the effect of artificial touch cues on orientation in microgravity. Neuroscience Letters, 404(1-2):78–82, 2006.

[7] Ravinder S Dahiya, Giorgio Metta, Maurizio Valle, and Giulio Sandini. Tactile sensingfrom humans to humanoids. IEEE transactions on robotics, 26(1):1–20, 2009.

[8] S. Tian, F. Ebert, D. Jayaraman, M. Mudigonda, C. Finn, R. Calandra, and S. Levine. Manipulation by feel: Touch-based control with deep predictive models. In 2019 International Conference on Robotics and Automation (ICRA), pages 818–824, 2019.

[9] Achu Wilson, Shaoxiong Wang, Branden Romero, and Edward Adelson. Design of a fully actuated robotic hand with multiple gelsight tactile sensors. arXiv preprint arXiv:2002.02474, 2020.

[10] Franka Emika. Franka emika panda robot and interface specifications, 2020. https://frankaemika.github.io/docs/control_parameters.html#.

[11] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. Journal of Robotics, 2012, 05 2012.

[12] Apex.AI. Apex.OS^{R}, 2020. https://www.apex.ai/apex-os.

[13] Open Source Robotics Foundation. Gazebo simulator, 2020. http://gazebosim.org/.

[14] David Hershberger, David Gossow, and Josh Faust. rviz, 2020. http://wiki.ros.org/rviz.

[15] Orocos kdl, 2020. https://www.orocos.org/wiki/orocos/kdl-wiki.

[16] Eigen, 2020. http://eigen.tuxfamily.org.

[17] ALGLIB Project. Alglib, 2020. https://www.alglib.net/.

[18] Open Source Robotics Foundation. Gazebo plugins, 2020. http://gazebosim.org/tutorials?tut=ros_gzplugins#Pluginsavailableingazebo_plugins.

[19] Josh Faust Dave Hershberger, David Gossow. rviz marker display type, 2020. http://wiki.ros.org/rviz/DisplayTypes/Marker.

[20] Richard S Hartenberg and Jacques Denavit. A kinematic notation for lower pair mechanisms based on matrices. Journal of applied mechanics, 77(2):215–221, 1955.

[21] justagist. Franka emika panda robot description, 2020. https://github.com/justagist/franka_panda_description.

[22] Robot state publisher ros package, 2020. wiki.ros.org/robot_state_publisher.

[23] José María Goicolea Ruigómez. Curso de Mecánica. 2010.

[24] David E. Orin and William W. Schrader. Efficient Computation of the Jacobian for Robot Manipulators. The International Journal of Robotics Research, 3(4):66–75, 1984.

[25] Bruno Siciliano. Kinematic control of redundant robot manipulators: A tutorial. Journal of intelligent and robotic systems, 3(3):201–212, 1990.

[26] Jim Hefferon. Linear algebra, 3rd.

[27] Tsuneo Yoshikawa. Dynamic manipulability of robot manipulators. Transactions of the Society of Instrument and Control Engineers, 21(9):970–975, 1985.

[28] Alain Liegeois et al. Automatic supervisory control of the configuration and behavior of multibody mechanisms. IEEE transactions on systems, man, and cybernetics, 7(12):868–871, 1977.

[29] Fabrizio Flacco, Torsten Kröger, Alessandro De Luca, and Oussama Khatib. A depth space approach to human-robot collision avoidance. In 2012 IEEE International Conference on Robotics and Automation, pages 338–345. IEEE, 2012.

[30] Fabrizio Flacco, Alessandro De Luca, and Oussama Khatib. Motion control of redundant robots under joint constraints: Saturation in the null space. In 2012 IEEE International Conference on Robotics and Automation, pages 285–292. IEEE, 2012.

[31] Yoshihiko Nakamura and Hideo Hanafusa. Inverse kinematic solutions with singularity robustness for robot manipulator control. 1986.

[32] Charles W Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. IEEE Transactions on Systems, Man, and Cybernetics, 16(1):93–101, 1986.

[33] Yitao Ding and Ulrike Thomas. Collision avoidance with proximity servoing for redundant serial robot manipulators.

[34] John Hughes (https://math.stackexchange.com/users/114036/john-hughes). Find a point on a line segment which is the closest to other point not on the line segment. Mathematics Stack Exchange. https://math.stackexchange.com/q/2193733.

[35] Sami Haddadin, Alessandro De Luca, and Alin Albu-Schäffer. Robot collisions: A survey on detection, isolation, and identification. IEEE Transactions on Robotics, 33(6):1292–1312, 2017.

[36] S. Haddadin, A. Albu-Schaffer, A. De Luca, and G. Hirzinger. Collision detection and reaction: A contribution to safe physical human-robot interaction. In 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 3356–3363, 2008.

[37] Numpy, 2020. https://numpy.org/.