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

needed?

needed?

Date

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.

After conclusions.

Conclusions smaller.

Often the abstract will be long enough to require more than one page, in which case the macro “\OnePageChapter” should *not* be used.

But this one isn’t, so it should.

Contents

Chapter 1

Introduction

1.1 Human-Robot Interaction

We are on the threshold of a new era in robotic technology that will change our lives in the way we live and work. This robotic revolution 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. (https://h2r.cs.brown.edu/about/)

One of the main fields of research working on the expansion of these abilities to interact with the physical world is human-robot interaction (HRI). HRI deals with the general problem of 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.
- Childcare robots will help parents with chores such as assisting at diaper changing or feeding, so that families can spend high-quality time together.
- Manufacturing robots will collaborate with people to assemble complex objects on reconfigurable assembly lines, increasing the efficiency and flexibility of factory floors.

The transition of robots from restricted access areas to human-populated environments comes with a number of problems that research needs to overcome, being safety one of the key issues that need to be addressed. Traditionally, safety in industrial robots has been achieved isolating them 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 ^{TM} 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 [1]. 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 [2]. 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 [3].

However, 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.
- **Dynamic tactile sensors.** If you press your finger against an objects corner, you can feel it for as long as you hold your finger in place. If you slowly rest your finger on the table, in contrast, you feel relatively little, until you start moving it gently back and forth. Suddenly, you are able to feel the texture, dustiness, scratches and a breadth of the properties the object has. These sensations are provided by the dynamic, fast acting tactile sensors that are embedded in the skin.
- **Detect changing light levels.** 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 ([4] [5]).

However, whole-body artificial skins have not gained the same attention as end-effector coverings. In fact, this type 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 compliance. Compliance is essential for interaction and extensive research has been conducted to deal with the issue, leading to two paradigms: **passive** and **active compliance**. Passive compliance aims to achieve safety through physical adaptation to the environment, by the use of flexible parts and mechanisms or by limiting capabilities such us speed. Active compliance, in contrast, relies on algorithms and control laws for reactive interaction with the environment. Both approaches are not mutually exclusive.

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. Active compliance makes use of not suitable sensors, which 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. This may be one of the reasons why none of the solutions have become the established standard in robotics.

Whole-body artificial skins have the potential to solve all of these issues, providing high-density relevant information, that is as close to the particular task of collision avoidance as it gets. At the same time, it can also be designed to be mechanically compliant to the environment.

Adaptable to the form of the robot.

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 adding proximity sensing capabilities to the skin opens up new possibilities in terms of safety robot control. The addition of proximity sensors allows a enables a compact, external motion capture system free system, with highly informative prior-to-contact data. This combined with lighter data, the main problems of depth-sensing are solved: occlusion is never again a problem and computational costs are greatly reduced.

1.4 Control and collision avoidance

From the control perspective, using proximity data for the safe movement of a robot might prevent performing the main task when any undesired object obstructs the desired trajectory. Nonetheless, this does not have to be the case if a redundant robot is used.

A robot is redundant if the number of degrees of freedom (DoF) n is bigger 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.

This means that safe motion 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

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. 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 3.2.1 we briefly present a framework for automatic kinematic calibration that leverages an IMU to accurately estimate the pose (position and orientation) of skin units along the surface of a robot. The main section is section 3.3, where the methods for collision avoidance are presented.

3.1 Required resources

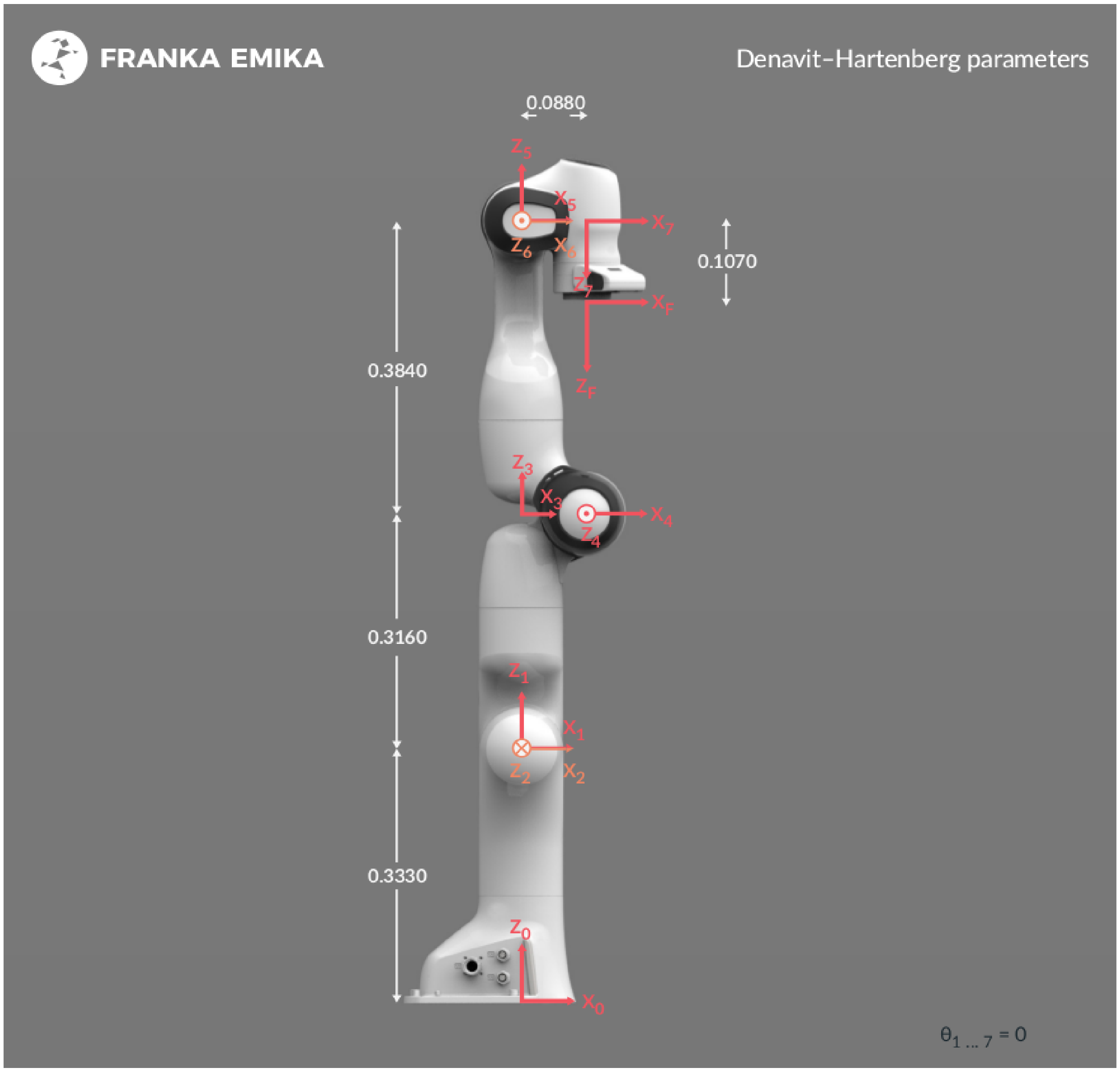
Introductory paragraph. General presentation of the required resources.

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 [6].



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 $\bar{\mathbf{c}}(\mathbf{q}, \dot{\mathbf{q}})$.

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 [6] and depicted in figure 3.1 are shown in table 3.1.

Table 3.1: Denavit Hartenberg parameters [6]

Joint	$\mathbf{a}$ (m)	$\mathbf{d}$ (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 [6]

	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	$\frac{\mathrm{rad}}{\mathrm{s}}$
$\mathbf{\ddot{q}_{max}}$	15	7.5	10	12.5	15	20	20	$\frac{\mathrm{rad}}{\mathrm{s}^2}$

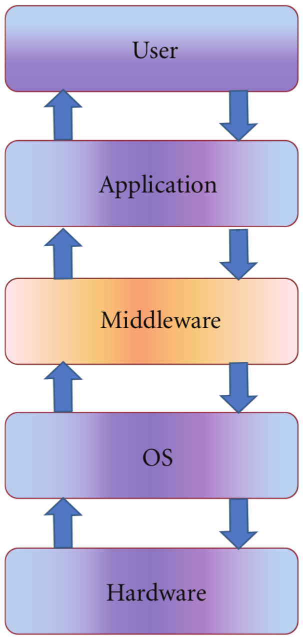
Table 3.3: Cartesian space limits [6]

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

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 [7] 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 [7]



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:

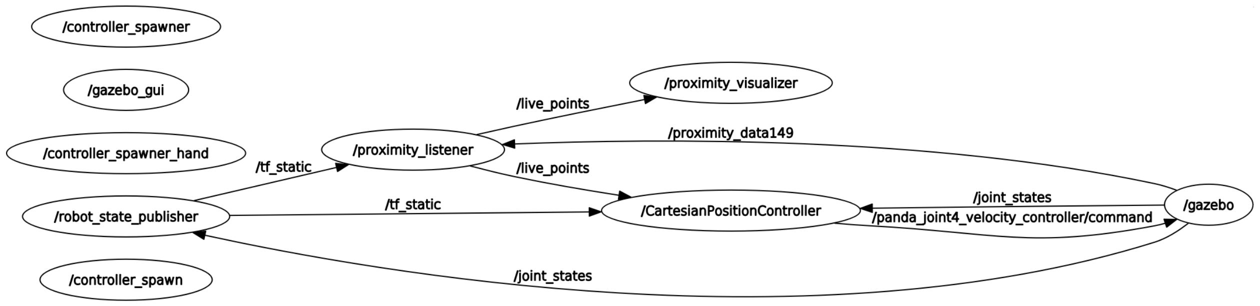
- **Topics.** A publisher node publishes a message to a topic. Another node subscribes to this topic, meaning that every time there is a new message published in the topic this subscriber node will receive it and therefore will we able to utilize and manipulate the data. Several nodes can publish to 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 [8].

3.1.3 Simulation and Visualization

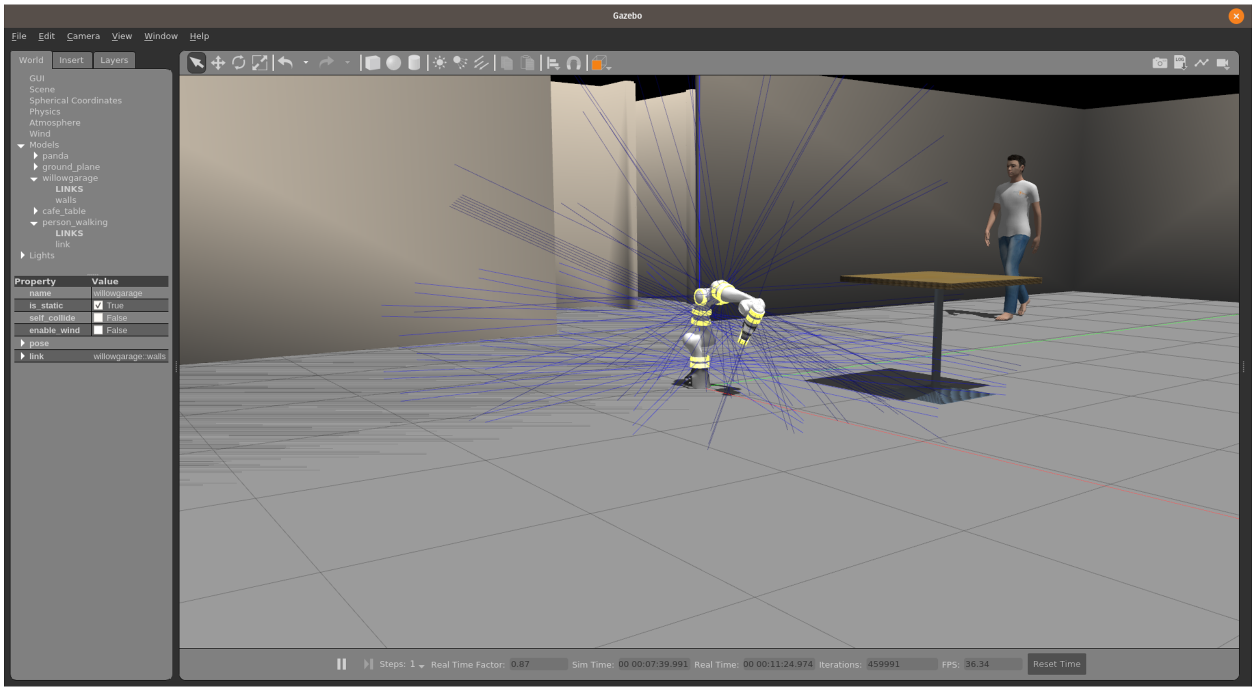
According to the Cambridge Dictionary, a simulation is a situation in which a particular set of conditions is created artificially in order to study or experience something that could exist in reality. Simulations are therefore very useful for research and especially for robotics, where the following 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 [9] 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.



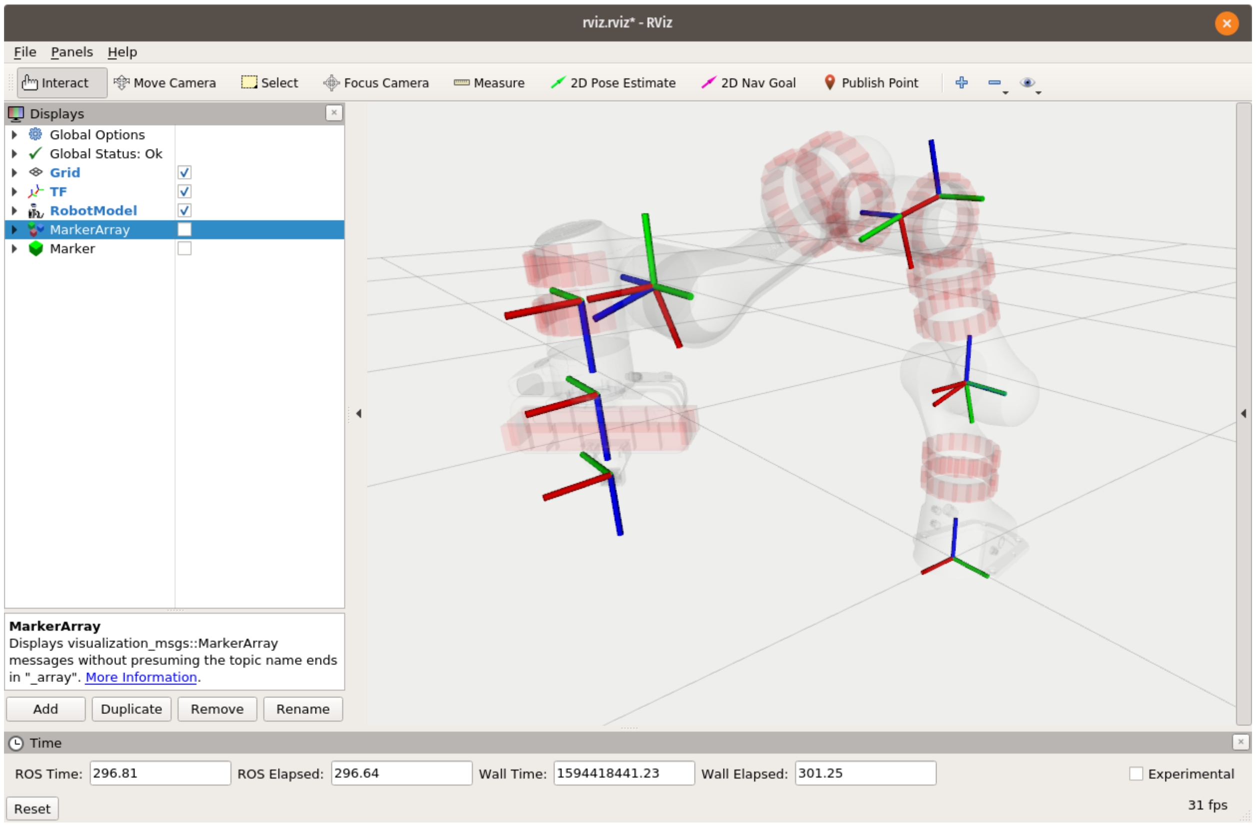
Property	Value
name	willowgarage
is_static	✓ True
self_collide	False
enable_wind	False
pose	
link	willowgarage::walls

```
Steps: 1  Real Time Factor: 0.87  Sim Time: 00 00:07:39.991  Real Time: 00 00:11:24.974  Iterations: 459991  FPS: 36.34
```

On the other hand, ROS offers a useful vizualization tool called *rviz* [10], 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*.



- Global Options
- Global Status: Ok
- Grid ✓
- TF ✓
- RobotModel ✓
- MarkerArray ☐
- Marker ☐

MarkerArray
Displays visualization_msgs::MarkerArray messages without presuming the topic name ends in "_array". More Information.

ROS Time: 296.81 ROS Elapsed: 296.64 Wall Time: 1594418441.23 Wall Elapsed: 301.25 ☐ Experimental

31 fps

3.1.4 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.

3.1.4.1 Robot Kinematics

The Kinematics and Dynamics Library (KDL) [11] 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.

3.1.4.2 Linear Algebra

Eigen [12] 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* (3.1.4.1).

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.

3.1.4.3 Optimization

ALGLIB [13] 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.

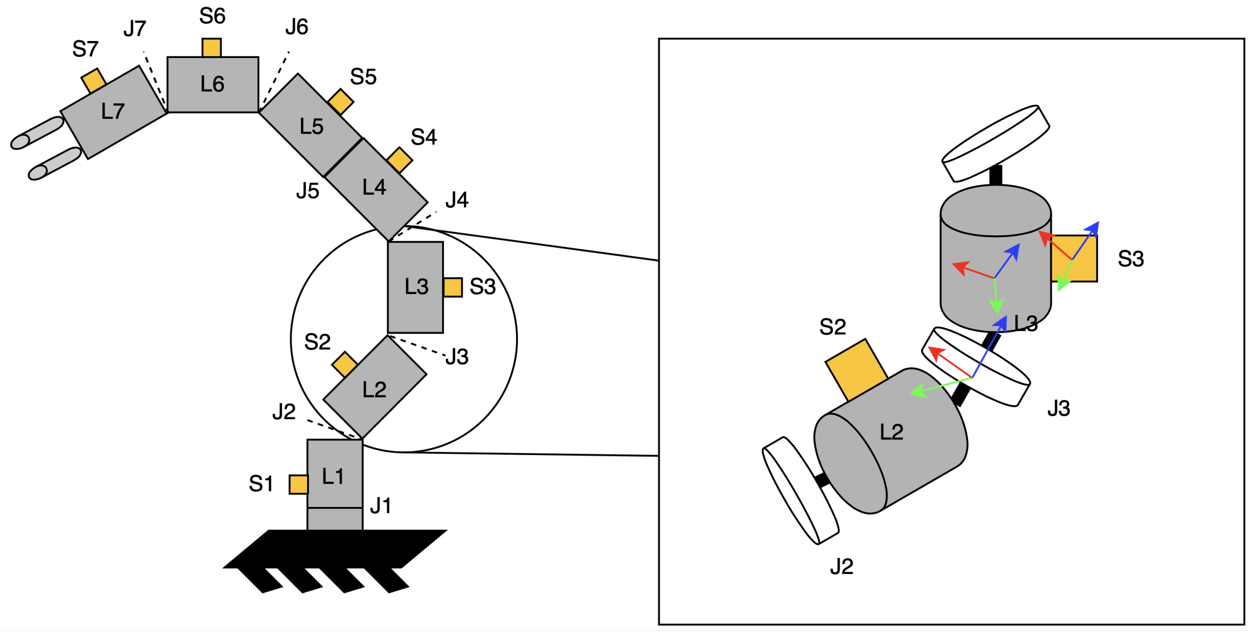
In this section we describe the method developed in this work to create a 3D point cloud from proximity distance data. The first step is the kinematic calibration of the skin units that contain the proximity sensors. Kinematic calibration makes an automatic estimation of the position and orientation 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 3.6. 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 3.6: 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.

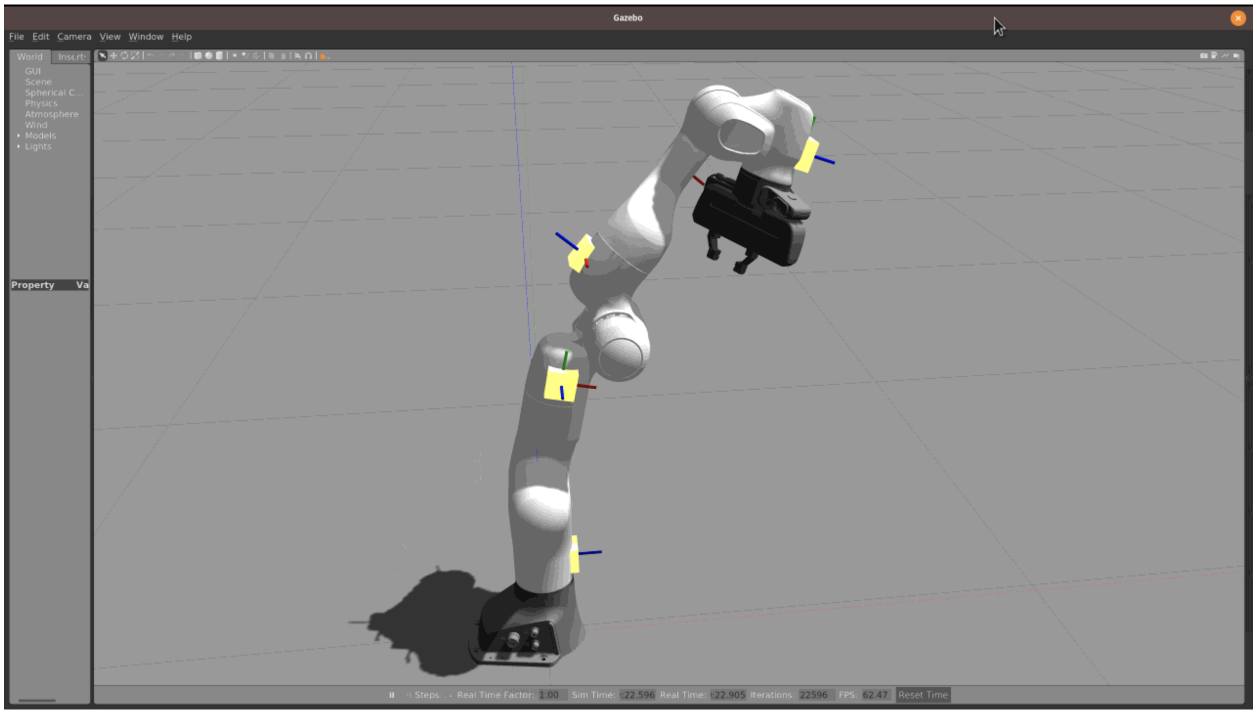
(4) **Minimize the error function with a global optimizer.** A global optimizer optimizes the DH parameters by minimizing a given error function. One example error function could be seen as the error between the measured accelerations from the IMUs and the estimated accelerations using the kinematic 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 3.7.. 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 3.7: Calibrated IMU positions on a simulated Franka Emika Panda robot.


Gazebo
File Edit Camera View Window Help
World Insert
GUI
Scene
Spherical C...
Physics
Atmosphere
Wind
Models
Lights
Property Va
Steps Real Time Factor 1.00 Sim Time 22.596 Real Time 22.905 Iterations 22596 FPS 62.47 Reset Time

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 [14], 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.

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \\ z &= z \end{aligned} \tag{3.1}$$

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 3.1.

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.

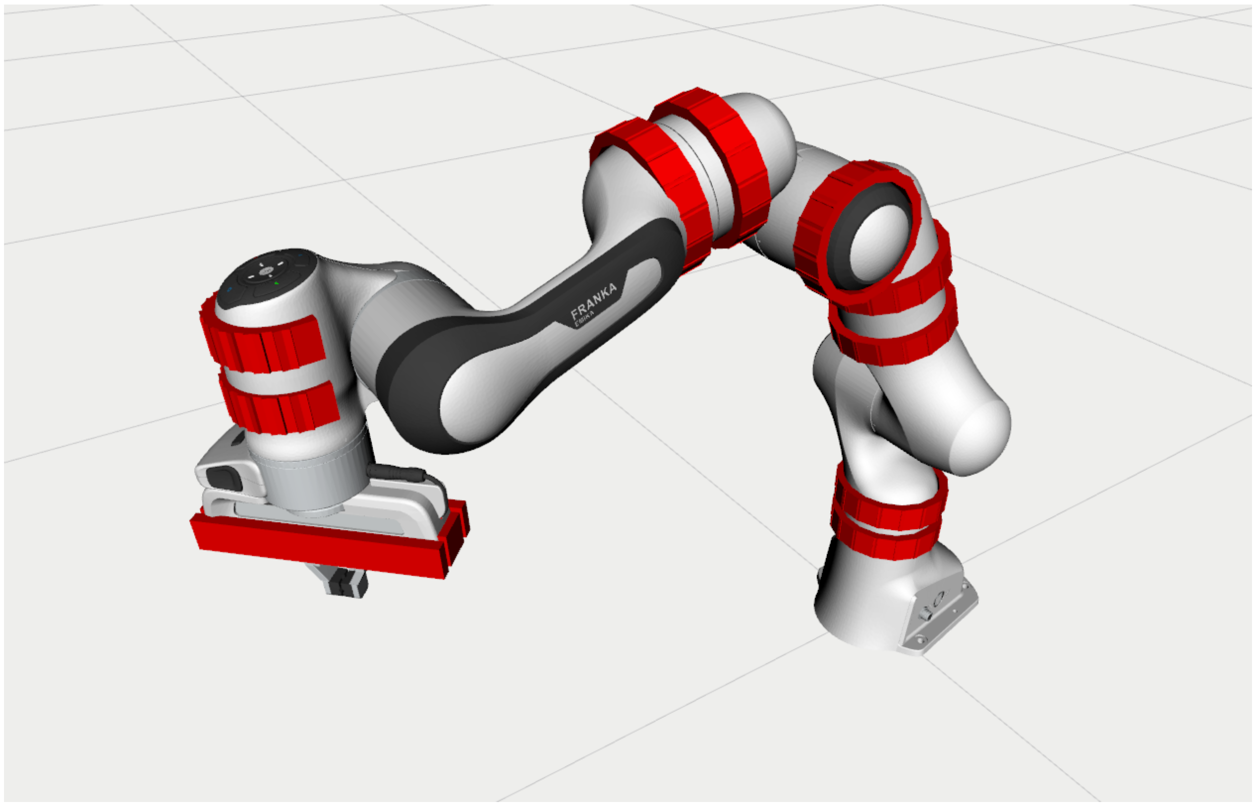
We can use this macro to place sensors through a number of rings to cover the body of the robot and simulate the artificial skin. `radius` specifies the selected link's radius and `z0` decides de longitudinal position in with we want to place the sensor. With `radius` and `z0` fixed, changing `theta` 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 3.8, where the new links are shown as red parallelepipeds. The final amount of sensors added for simulation counts 162.

Figure 3.8: Robot body covered with proximity sensors, which appear in red.

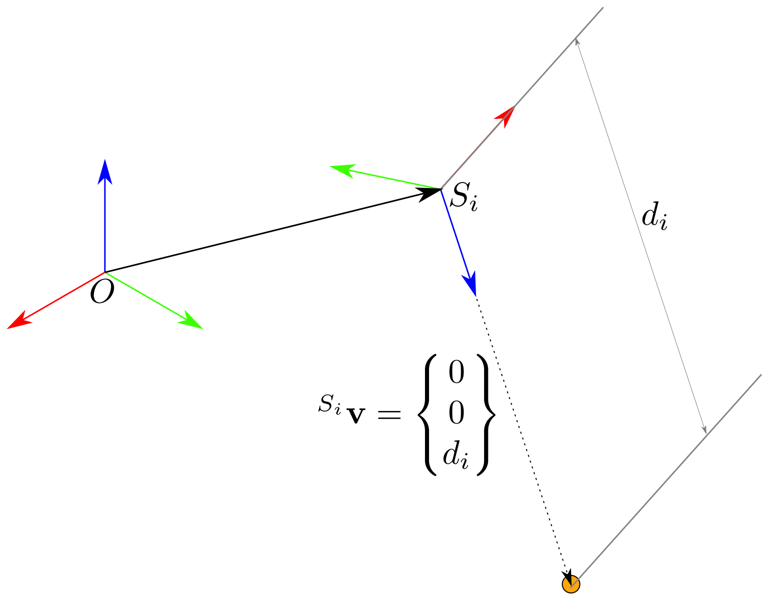


3.2.3 Point localization algorithm

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 3.9 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 3.9: Diagram showing the conversion of a proximity sensor reading into a 3D point in space.


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

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).

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

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 3.9). 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:

$$^{0}\mathbf{v} =^{0} \mathbf{p}_{S_i} +^{0} \mathbf{R}_{S_i} \, ^{S_i}\mathbf{v} \tag{3.2}$$

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
```

```
 5      (
 6          scan->header.frame_id.substr
 7          (
 8              scan->header.frame_id.find_first_of("0123456789"),
 9              scan->header.frame_id.length() -1
10          )
11      );
12
13  try
14  {
15      listener.lookupTransform
16      (
17          "/world",
18          "/proximity_link" + std::to_string(sensor_number),
19          ros::Time(0),
20          transform[sensor_number]
21      );
22      translation1[sensor_number] <<
23          transform[sensor_number].getOrigin().getX(),
24          transform[sensor_number].getOrigin().getY(),
25          transform[sensor_number].getOrigin().getZ();
26      translation2[sensor_number] << 0.0,
27                                     0.0,
28                                     scan->ranges[0];
29      rotation[sensor_number].w() =
30          transform[sensor_number].getRotation().getW();
31      rotation[sensor_number].x() =
32          transform[sensor_number].getRotation().getX();
33      rotation[sensor_number].y() =
34          transform[sensor_number].getRotation().getY();
35      rotation[sensor_number].z() =
36          transform[sensor_number].getRotation().getZ();
37
38      live_points[sensor_number] =
39          translation1[sensor_number] +
40          (rotation[sensor_number] * translation2[sensor_number]);
41  }
42  catch (tf::TransformException ex)
43  {
44      ROS_ERROR("%s",ex.what());
45      ros::Duration(1.0).sleep();
46  }
47  }
```

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

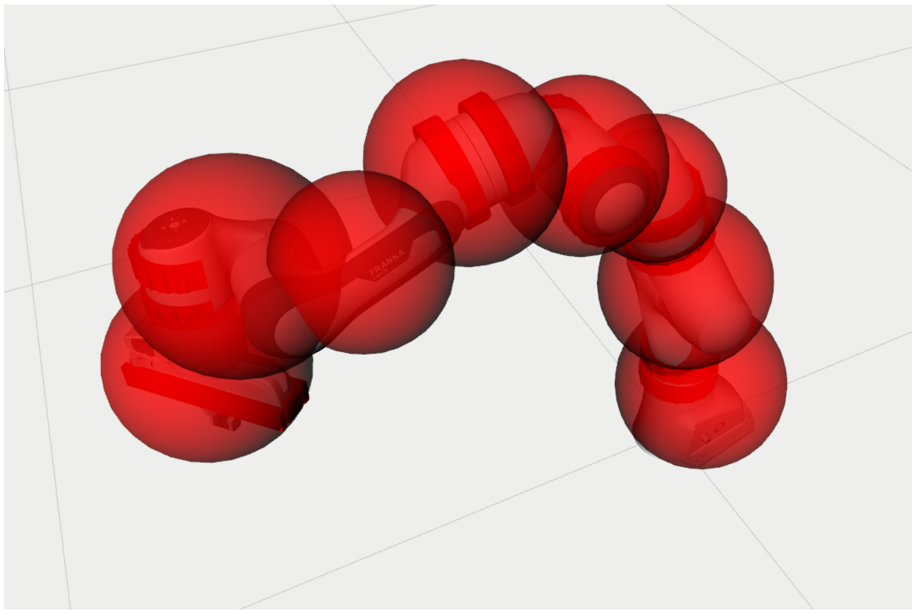
The first thing the callback function does is read the sensor from which it is receiving the message. This is done in line 3. There is a single callback for all the sensors, so this step is crucial.

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 3.2. 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 3.10, and checking if the points are inside those spheres. This is done in `isInSphere` in line 10 of listing 3.6.

Figure 3.10: 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::Points`. This is done at 50 Hz, or what is the same, every 20 ms.

BUFFER IS LEFT

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 Arrays` [15] 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 3.7, 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 3.7: 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 3.8.

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.

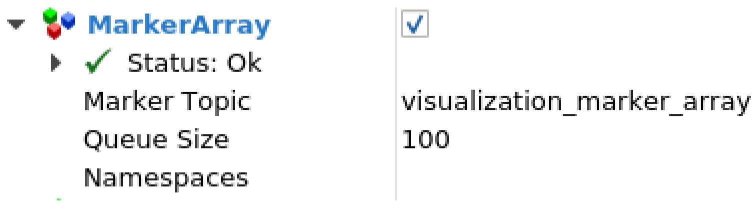
```
    void ProximityVisualizer::Callback(
        const ros_robotic_skin::PointArray::ConstPtr& msg)
    {
        // Delete all points from previous callback
        marker.action = visualization_msgs::Marker::DELETEALL;
        marker_array.markers.push_back(marker);
        pub.publish<visualization_msgs::MarkerArray>(marker_array);
        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 3.8: 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 3.11. Since the `Namespaces` field is empty, all the points in the received `MarkerArray` will be visualized, with no exception.

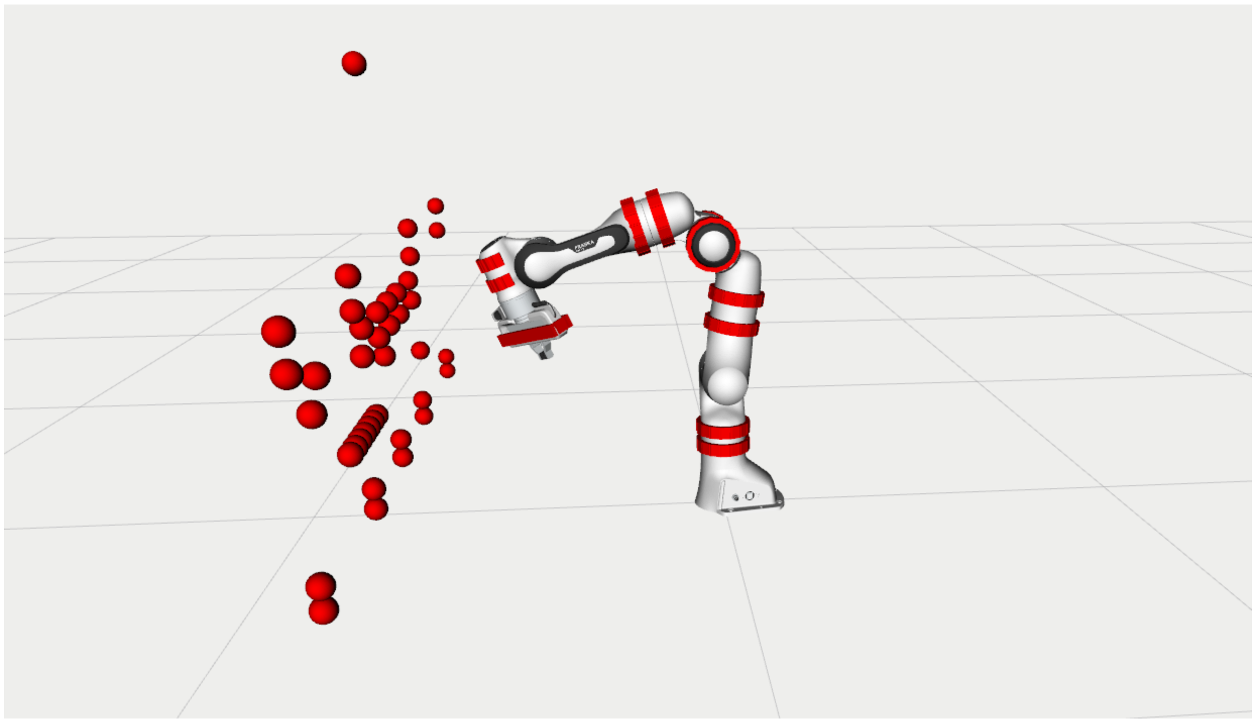
Figure 3.11: Menu of the Marker Array plugin in *rviz*.



Marker Topic	visualization_marker_array
Queue Size	100
Namespaces	

The points are seen in *rviz* as shown in figure 3.12, where a wall in front of the robot is being perceived.

Figure 3.12: Final visualization of the perceived points in *rviz*.



3.3 Control

INTRODUCTORY PARAGRAPH

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 [16]. 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 [6].

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 [17]) Emika describes the transformation from frame 6 to frame 7.

```
<joint name="${arm_id}_joint7" type="revolute">
      <safety_controller k_position="100.0"
                         k_velocity="40.0"
                         soft_lower_limit="-2.8973"
                         soft_upper_limit="2.8973"/>
      <origin rpy="${pi/2}␣0␣0" xyz="0.088␣0␣0"/>
      <parent link="${arm_id}_link6"/>
      <child link="${arm_id}_link7"/>
      <axis xyz="0␣0␣1"/>
      <limit effort="12" lower="-2.8973"
```

```
                              upper="2.8973"
                              velocity="2.6100"/>
     <dynamics damping="1.0" friction="0.5"/>
</joint>
```

Listing 3.9: 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 3.3.

$$ {}^{i-1}\mathbf{T}_i = \begin{pmatrix} {}^{i-1}\mathbf{R}_i & {}^{i-1}\boldsymbol{p}_i \\ \mathbf{0}^{\mathrm{T}} & 1 \end{pmatrix} \tag{3.3} $$

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 $i = 1, \dots, n$, the full transformation from the base $i = 0$ to the end effector $i = n$ transformation can easily be calculated:

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

Forward Kinematics are calculated in the robot_state_publisher package [18]. 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:

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

It is trivial to see that solving the position ${}^0\boldsymbol{p}_n(q_1, \dots, q_n) = ({}^0\boldsymbol{p}_n)^d$ 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 $\mathbf{q} = \begin{bmatrix} q_1 & \ldots & q_n \end{bmatrix}^\intercal$ 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:

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

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 $\dot{\mathbf{t}}(t)$,

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

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 [19].

(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:

$$\mathbf{v}_P = \mathbf{v}_O + \boldsymbol{\omega} \times \overrightarrow{\mathbf{OP}} \tag{3.4}$$

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:

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

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

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

$$\dot{\mathbf{x}} = \mathbf{J}(\mathbf{q}) \cdot \dot{\mathbf{q}} \tag{3.5}$$

There are many methods that compute the values of the jacobian matrix $\mathbf{J}(\mathbf{q})$ [20]. The general idea underlying is similar in all of them, with differences in efficiency.

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

$\mathcal{J}_{Pi} \in \mathbb{R}^{3\times1}$ denotes the contribution of $\dot{q}_i$ per radian to $\mathbf{v}_{\text{end effector}}$. $\mathcal{J}_{Oi} \in \mathbb{R}^{3\times1}$ 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 3.4. It is obvious that the contribution of a prismatic joint to $\boldsymbol{\omega}_{\text{end effector}}$ is $\mathbf{0}$.

KDL

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 [21], 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 3.5. 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:

$$\underset{m\times n}{\mathbf{J}(\mathbf{q})} \cdot \underset{n\times 1}{\dot{\mathbf{q}}} = \underset{m\times 1}{\dot{\mathbf{x}}_{\mathbf{d}}} \tag{3.6}$$

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 3.6 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 [22]. 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, by 6, 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 $<=$ than this maximum rank. In the case of $\mathbf{J}(\mathbf{q})$ in a redundant robot this is given by the dimension of the row space m.

Theorem 5. *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 $(n\text{-}r)$

are equivalent.

The homogeneous system associated to equation 3.6, $\mathbf{J}(\mathbf{q})\cdot\dot{\mathbf{q}} = \mathbf{0}$, consequently 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: $(\mathrm{n} - \mathrm{r}) \geq (\mathrm{n} - \max(\mathrm{m})) = (7 - 6)$. **THIS RESULTS IN THE FACT THAT INFINITE SOLUTIONS**.

PROVE THAT IF $r < m$ no particular solution exists.

Therefore when the rank of $\mathbf{J}(\mathbf{q})$ becomes smaller than m, the ability to control one axis either of translation or rotation is lost. The positions q in which this situation is given are called singularities.

Summing up, the problem has been defined as a set of linear equations (equation 3.6) and the existance of the solutions has been studied until now. Next, we must find a solution to the set of equations.

GENERAL SOLUTION OF THE EQUATION

Pseudoinverse

However, doesn't avoid singularities.

NUMERIC INSTABILITY SINGULARITIES OF PSEUDOINVERSE The reader may think that it would be useful to have a measure of when one of this situations is about to take place. [23] introduced a measure that indicates how close a robot is from a singularity:

$$\sqrt{\mathbf{J}^{\intercal}\cdot\mathbf{J}}$$

This value tends to 0 when the vector q tends to a position in which the rank of $\mathbf{J}$ will be decreased.

Leageouis and middle range of the joints

3.3.1.6 End effector Position Control through Inverse Instantaneous Kinematics

```cpp
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);
    qRanges = jointLimitsMax - jointLimitsMin;
    return 2.0/7.0 * (q - qMid).cwiseQuotient(qRanges);
}
```

Listing 3.10: CartesianPositionController.cpp

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

3.3.3 Optimization based collision avoidance algorithm

Chapter 4

Results

youtube links, charts... from flacco, QP simulations and real robot if posible.

illustrations, block diagrams (http://dia-installer.de/) powerpoint

Wait for this last part.

Chapter 5

Conclusions

introduction summary

one short paragraph about

results

future work

Bibliography

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

[2] 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.

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

[4] 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.

[5] 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.

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

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

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

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

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

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

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

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

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

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

[16] 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.

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

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

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

[20] 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.

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

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

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

[24] 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.

[25] 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.

[26] Numpy, 2020. `https://numpy.org/`.