Establishment of software architecture towards integrated simulation of human-robot collaboration

A thesis presented for the degree of
BSc in Engineering (Robot Systems)

written by

Martin Androvich
(marta16@student.sdu.dk)

under the supervision of

Cheng Fang
(chfa@mmmi.sdu.dk)

University of Southern Denmark (SDU)
Technical Faculty (Faculty of Engineering)

August, 2020

Abstract

For research in safety of Human-Robot Interaction and Collaboration (HRIC), an integrated simulation environment is desired, in which a software architecture must facilitate the means of real time robot interfacing and collection of sensory data, furthermore providing a generic framework for development and deployment of an arbitrary robot controller in both a simulated and real environment.

Using the workcell in SDU Industry 4.0 lab, equipped with a Franka Emika Panda robot and various perceptual sensory equipment (e.g., motion capture system, EMG sensors etc.), the Robot Operating System (ROS) middleware is proposed as the underlying framework of the software architecture, in which the development of the generic controller framework is be accommodated by ROS Control and Franka ROS, using Gazebo as the simulation environment.

A simple marker tracking experiment is used to evaluate the capabilities of the software architecture, proving to be adequate for both real time interfacing of the robot and online data collection.

The integration of the Panda into Gazebo requires a kinematic robot model, estimated dynamic parameters and an emulated hardware interface for the joint space robot dynamics, which are necessary for any form of dynamics control. Such integration is evaluated using a joint space PD controller with gravity compensation, which displays similar joint movement to that of the real robot; presumably improvable with tuned PD values and more precise dynamics parameters.

To analyze human-robot interaction, the robot model is integrated into a biomechanical simulator (OpenSim), automating the transformation of a Unified Robot Description Format (URDF) model to OpenSim Model Format (OSIM). A similar PD controller is implemented in OpenSim, albeit with modifications, exhibiting similar joint positions when compared to its, also modified, counterpart in Gazebo.

Contents

Glossary

Acronyms

cobot collaborative robot.
CoM center of mass.
DLL dynamic-link library.
DoF degrees of freedom.
FCI Franka Control Interface.
HBP Human Brain Project.
HRIC Human-Robot Interaction and Collaboration.
KDL Orocos Kinematics and Dynamics Library.
MOCAP motion capture system.
NRP Neurorobotics Platform.
OSIM OpenSim Model Format.
ROS Robot Operating System.
SDK software development kit.
SDU University of Southern Denmark.
URDF Unified Robot Description Format.

Terms

collaborative robot (cobot) Robots intended for direct human robot interaction within a shared space or where humans and robots are in close proximity.

degenerate In mathematics, a degenerate case is a limiting case of a class of objects which appears to be qualitatively different from (and usually simpler to) the rest of the class.

middleware Software that connects software components or applications, typically being the layer between the operating system and user applications.

Panda The Franka Emika Panda robot.

robot dynamics The joint space robot dynamics for a given robot state, described by the mass matrix, $\mathbf{M}(\boldsymbol{q})$, Coriolis matrix, $\mathbf{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$, and gravity vector, $\boldsymbol{g}(\boldsymbol{q})$.

robot state The state of the robot in terms of joint space position, $\boldsymbol{q}$, and velocity, $\dot{\boldsymbol{q}}$.

rotatum The derivative of torque with respect to time, i.e., the rate of torque.

Chapter 1

Introduction

1.1 Motivation

In industry, collaborative robots (cobots) are employed as intelligent partners which are able to work closely together with human workers to collaboratively perform complicated tasks in production lines. The safe and harmonic co-existence of cobots and human coworkers in a shared workspace without any safety guards is implemented by low-level control mechanisms, which impose kinematic and dynamic constraints (speed and force) and protective stops on cobots in risky cases (e.g., Universal Robots). This safety strategy aims to make cobots avoid unintentional contacts with humans or reduce the risk of harming humans when intentional contacts are required. Therefore, the nature of this strategy can only enable cobots and humans to work collaboratively in sequence for a series of process steps in a complicated task without continuous contact with each other (occasional contact may be needed, e.g., handover operation). Accordingly, continuous human-robot collaboration tasks have been rarely seen in industry even though typical scenarios such as, human-robot co-transportation and co-assembly tasks, have been discussed extensively in academia. One of the important and intractable issues is that the safety and ergonomics cannot be verified and validated in continuous human-robot collaboration tasks.

When a human coworker and a cobot manipulate on a common heavy and bulky object (e.g., a car engine), it is risky to validate the safety of the whole process through actual experiments since any accidents in the experiments can seriously hurt the human coworker, for instance, the abnormal behaviors of the designed controller of the cobot can generate large adverse force on the human and cause joint injuries or make the object fall down and hurt his/her foot. Therefore, it is crucial to assess the level of the risk of a proposed human-robot collaborative task through a large number of high-fidelity simulations before any actual experiments. To this end, an integrated and unified simulation environment for robots and humans has to be established as an important and necessary preparation for the demanding human-robot collaboration simulations.

Nowadays, the most popular simulation environment in robotics community is Robot Operating System (ROS) while its counterpart in human biomechanics community is OpenSim. Currently, these two simulation environments are rather separate and isolated despite of the fact that the two communities are increasingly collaborating with each other since more and more attention is being drawn in the topic of human-robot interaction/collaboration.

1.2 Project description

A Human-Robot Interaction and Collaboration (HRIC) workcell has been established in Industry 4.0 lab at the Maersk Mc Kinney Moller Institute. This workcell consists of cobots, namely the Franka Emika Panda, and various measurement devices (e.g., force plate), as shown in figure 1.

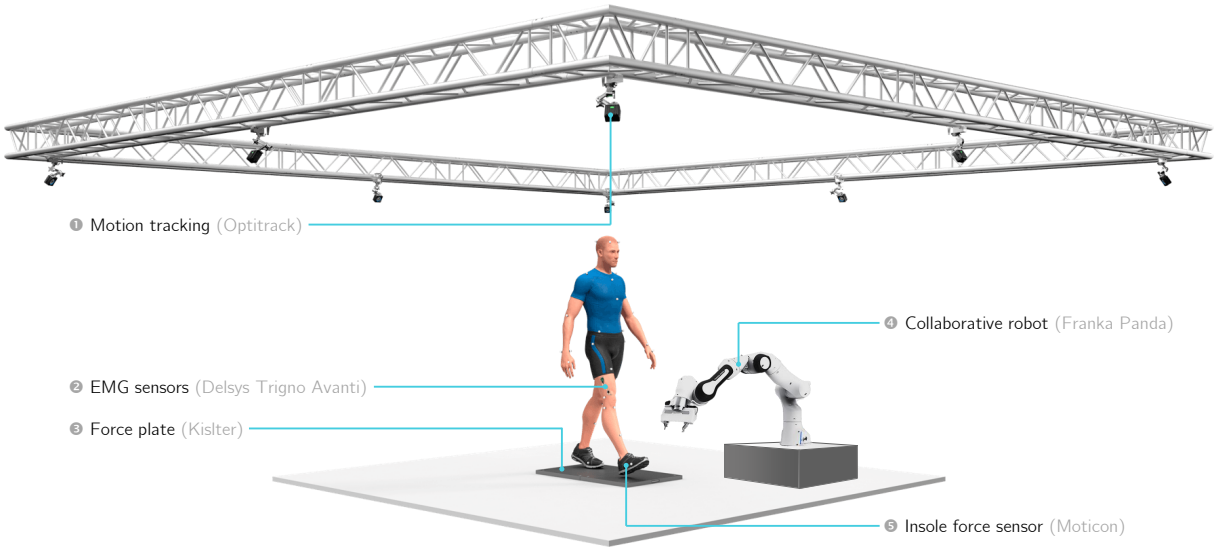


Figure 1: The setup of a Human-Robot Interaction and Collaboration workcell in SDU Industry 4.0 lab.

In this workcell the overall aim is to establish a safety and ergonomics benchmark by developing an integrated simulation environment and using the physical platform for data collection and analysis of continuous human-robot collaboration tasks. The general idea is to simulate a robot with an arbitrary controller in contact with a musculoskeletal model of the human body (e.g., a human arm), allowing to evaluate the bio-mechanical effects of their interaction and use that data to perform an assessment of safety and risk levels - here the physical platform is used to verify the results. By using such benchmark, the end-users are supposed to get an idea of whether the proposed human-robot collaboration solutions are safe and deployable.

The objective of this project is to propose a software architecture that facilitates both (a) the development of an integrated simulation environment and (b) the usage of the physical platform for data collection. Specifically, the software architecture should permit real-time robot interfacing and the usage of external sensors for synchronous data-collection; the architecture must also enable the simulation of a robot with a generic controller, meaning that an arbitrary controller will exhibit similar behavior in both reality and simulation.

Finally, it must be researched whether the proposed software architecture may be extended to accommodate implementation of robot simulations in a biomechanical simulator (specifically OpenSim), such that the employed generic robot controllers are also platform (simulator) agnostic.

Chapter 2

Software architecture

2.1 Overview

The purpose of a software architecture is to define the underlying structure of a software system, essentially providing both the building blocks and guidelines for a unified software development process. Robotic systems are often complex and span several domains, such as hardware abstraction, low-level device control, control algorithms, sensing and data processing, inter-node communication and so forth. Establishing a modular, component-based software framework with reusable components and a common communication pipeline allows for a greatly simplified development process and enables better project manageability and faster prototyping, which is highly advantageous, especially as projects grow in size; such frameworks are commonly referred to as middleware.

There are various middleware solutions that provide a framework for a modular development of robotics systems, one of which is the Robot Operating System (ROS). It is a state-of-the-art of robotic framework and tends to be a standard for robotic application development [1]. ROS provides seamless integration with the widespread Gazebo 3D simulator and the Franka Emika Panda has a ROS software package for real-time robot interfacing, making ROS a good option for the requirements of this project.

2.2 Robot Operating System (ROS)

ROS [2] is an open-source, meta-operating system, meaning it runs on top of e.g. Linux and provides necessary, platform independent services for robot interfacing; most importantly, ROS provides a communication infrastructure with a package management and build system system (catkin) [3] that enables code modularity and reusability.

2.2.1 Infrastructure

ROS is a distributed framework of nodes, which are independent executables that can be loosely coupled at runtime; each node has a specific task it executes, e.g., process camera data or perform motion planning.

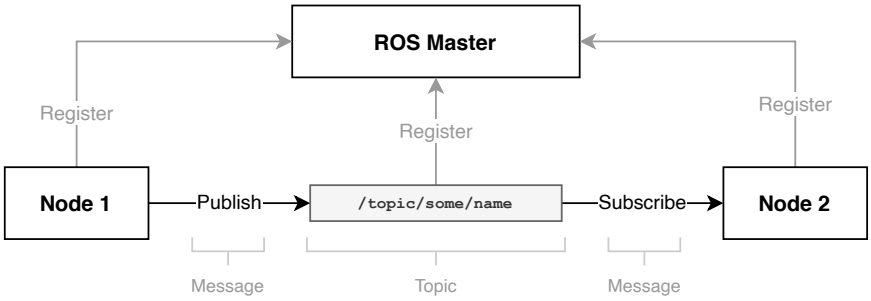


Figure 2: Diagram of ROS communication structure.

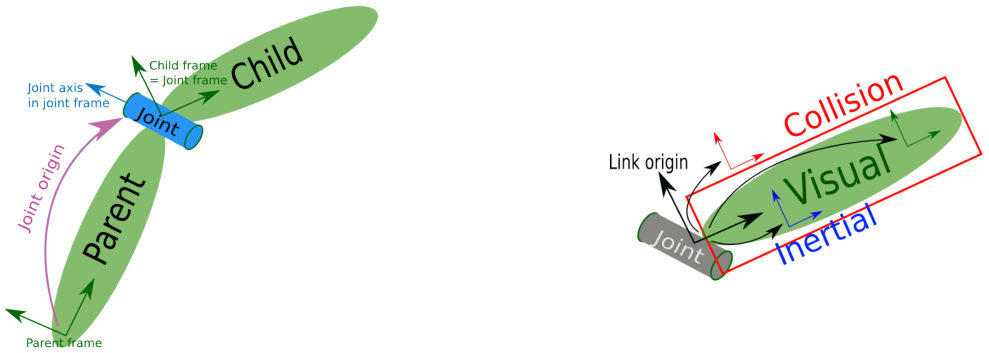
The primary form of communication in ROS consists of messages and topics, as illustrated in figure 2. Messages are predefined data structures and topics are named buses over which nodes exchange such messages; the ROS master server acts as a nameserver for the registered nodes and topics.

Topics have anonymous publish/subscribe semantics and are indented for unidirectional and asynchronous communication. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic.

Several nodes can be bundled into a package, containing any necessary definitions, libraries or anything else that logically constitutes a useful module. Packages are maintained using `catkin` build system [3], which is an extension of CMake, where such packages are typically organized in workspaces. Nodes can be executed individually or launched simultaneously in a preconfigured manner using the `roslaunch` tool.

2.2.2 Robot description (URDF)

The Unified Robot Description Format (URDF) is an XML format for representing a robot model, i.e., describing the kinematic, visual, collision and inertial (dynamic) properties of a robot.



(a) Frames defining links and joints.

(b) Properties of a robot link.

Figure 3

The description of a robot consists of a set of link elements, and a set of joint elements connecting the links together, as shown in 3a. Each link has a visual, collision and inertial properties and is typically defined by two frames: a link origin frame and a link inertial frame, as shown in figure 3b.

Link origin frame
This frame is specified by the joint that connects the child link to it's parent link, i.e. the transform from parent link to child link.

Link inertial frame
The frame to be set at the center of gravity relative to the link origin frame, defining inertial properties for the link.

Joints connect two links: a parent link and a child link. The transformation between the links is defined via the joint origin property, with the joint being located at the origin of the child link, ergo the joint frame equals the child link origin frame (when the joint variable is zero).

For any given joint, the axis of motion (unit vector) is also specified in the joint frame of reference; this is the axis of rotation for revolute joints or the axis of translation for prismatic joints.

2.3 Structure

The Human-Robot Interaction and Collaboration (HRIC) software architecture is structured as a decentralized system of ROS nodes, grouped for brevity, as illustrated in figure 4.

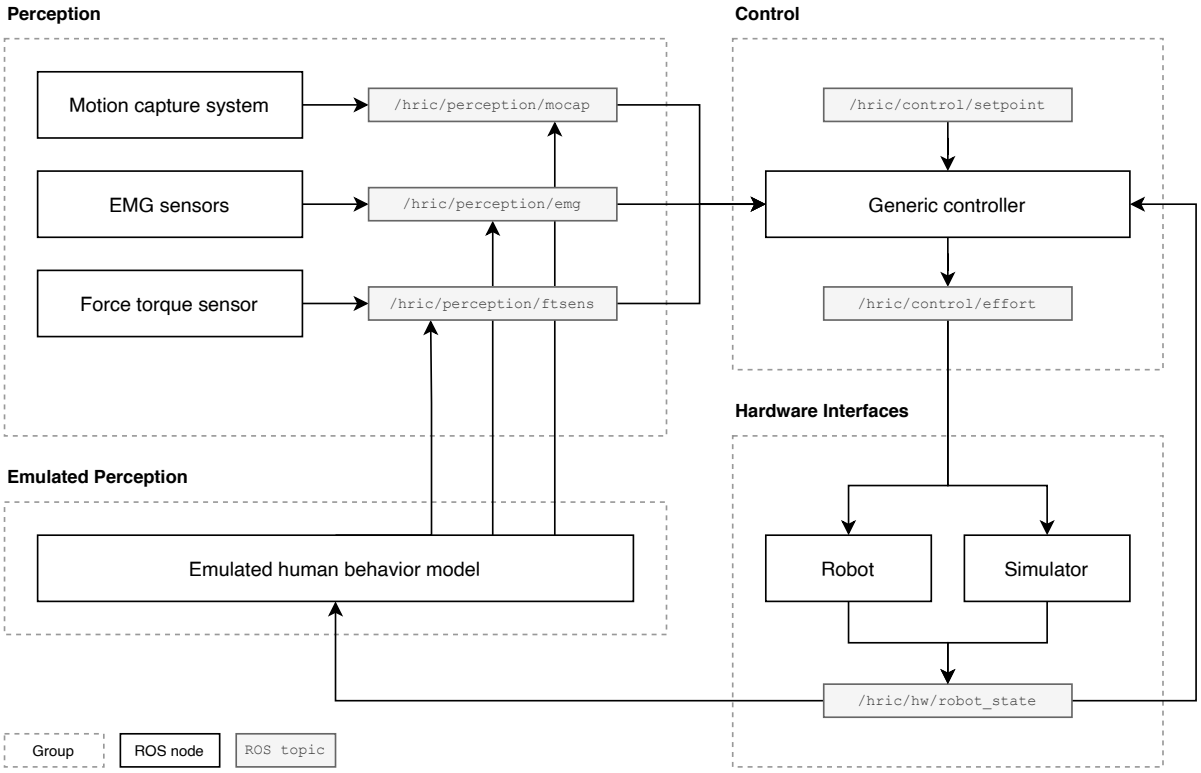


Figure 4: Proposed ROS-based software architecture.

The processes of the HRIC system, such as the motion capture system, are made modular by being implemented as ROS nodes and bundles as packages, containing containing message definitions, libraries and so forth. The processes are grouped, albeit being a purely virtual abstraction. For example, the motion capture system is part of the *perception* group.

Communication between nodes is achieved using ROS messages and topics, where each node constructs its data using message definitions and publishes the data to a topic; such topics are systematically named, prefixed by a common namespace (in this case `/hric`) and then their respective groups (e.g. `/perception`). For example, the motion capture system node would publish its data to the topic `/hric/perception/mocap` using the `msg/MocapData` message definition.

Other nodes need only know the topic name and include the necessary packages in order to access the custom message definitions and thus being able to subscribe to the published data.

For the purpose of data collection, a single node can be set to subscribe to various topics (e.g., different sensors), creating a centralized data collection node; even if the data is sampled at different frequencies, the `message_filter` [4] ROS package enables a unified, synchronous callback. Furthermore, different ROS packages can be used to process the data via the ROS communication framework (topics), such as the PlotJuggler [5] package.

The *perception* group is comprised of processes related to sensory input to the system and are implemented as nodes that sample, structure and publish such data. This primarily entails any human perception processes, such as muscle activity, applied force and so forth.

The *hardware interfaces* group uses the `ros_control` [6] package to provide a generic framework for robot interfacing and control, exposing common hardware interfaces, which in combination with Gazebo allows the same control signal to be sent to either the real or simulated robot [7], which in turn feedbacks the robot state.

This framework enables the *control* group, where the actual control loop (e.g., PID controller) is implemented as a ROS node; it can subscribe to an input setpoint topic, various sensory data topics and has real-time access to the robot state.

Furthermore, the *emulated perception* group can be added in order to emulate perceptual data when a simulation environment is used and no real sensory data is available; by using the robot state, an emulated human behavior model can be implemented to produce data in response to the robot states during human-robot interaction, e.g., data similar to the motion capture system (using the same format) would be published to the same topic and used interchangeably by the controllers, i.e., not requiring any further changes.

Chapter 3

Real time interfacing and data collection

3.1 Overview

Both ROS and Franka provide various tools that enable interfacing of the robot; these tools must be integrated into the proposed, ROS based software architecture, evaluating whether the system is capable of real-time interfacing and data collection. Such capabilities are verified by a simple application in which the robot end effector tracks the relative position changes of a motion capture system (MOCAP) marker whilst visualizing such data.

3.2 Robot interfacing

The Panda robot system is composed of a robot arm, hand and a control device. The Franka Control Interface (FCI) [8] enables a direct low-level bidirectional connection to the Panda robot, as illustrated in Figure 5, providing the robot state $[\boldsymbol{q}, \dot{\boldsymbol{q}}]$ and the joint-space robot dynamics $[\mathbf{M}(\boldsymbol{q}), \mathbf{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}), \boldsymbol{g}(\boldsymbol{q})]$, and enabling direct control via an external workstation PC connected to the robot via Ethernet.

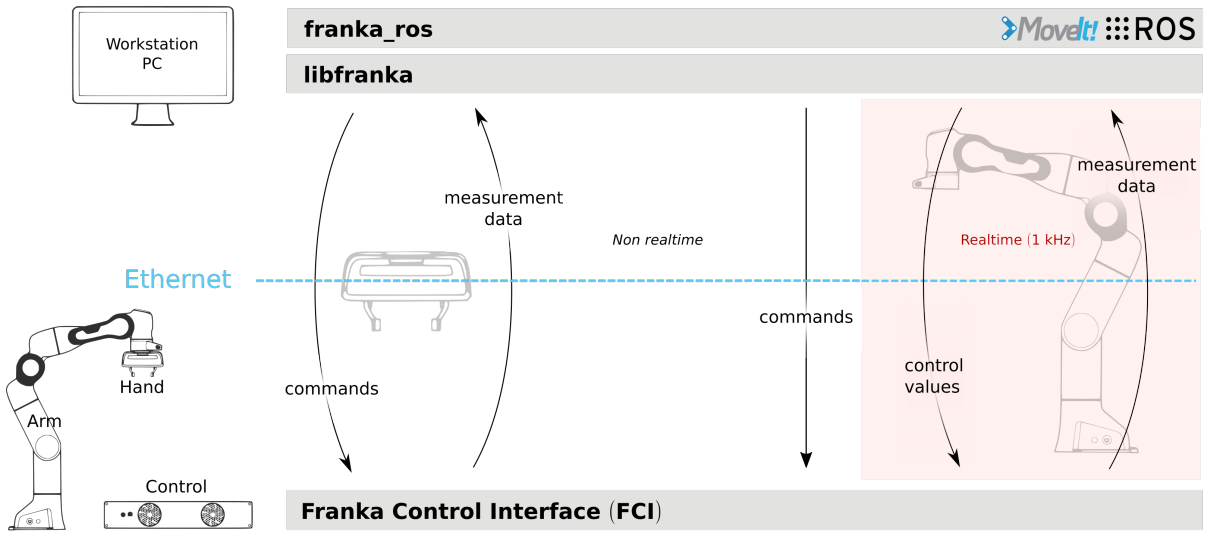


Figure 5: Schematic overview of FCI architecture.

The `libfranka` library, which is a `C++` implementation of the client side of the FCI, handles network communication with the control device and provides real time 1 kHz control interfaces to:

- Gravity & friction compensated joint level torque commands
- Joint position or velocity commands
- Cartesian pose or velocity commands

Furthermore, `libfranka` also gives access to real time, 1 kHz measurements of measured joint data, estimation of externally applied torques and wrenches and various collision and contact information. It also provides access to a robot model library, including forward kinematics, Jacobian matrices and robot dynamics (inertia matrix, Coriolis matrix, centrifugal vector, gravity vector) of all robot joints.

Franka provides the `franka_ros` metapackage, which includes various packages that integrate `libfranka` into the ROS ecosystem, providing a high-level interface to the Panda robot, including support for ROS Control and MoveIt!.

3.2.1 System configuration

The FCI has strict requirements [9] for both the workstation that interfaces with the robot and the network over which the communication is performed. The operating system should leverage real time capabilities (e.g. Linux patched with a real time (`PREEMPT_RT`) kernel) and the network card and cables used must support the Fast Ethernet standard, i.e. being able to carry traffic at the nominal rate of 100 Mbit/s.

The workstation system is running a Ubuntu (`18.04.3 LTS`) distribution which is patched with a real-time kernel (`5.4.10rt`) and configured to have the necessary real time capabilities with furthermore having optimized performance settings. The setup and configuration process is rather error prone and has therefore been automated using bash scripts, allowing to quickly configure a fresh install of Ubuntu.

The installation of the ROS framework and `libfranka` is likewise automated using a bash script. The project itself is hosted on GitHub [], being configured as a catkin workspace, which allows to build multiple, interdependent packages contained in a common workspace (i.e., directory). Furthermore, `rosdep` [] is used for package dependency management, meaning that the HRIC project can easily be configured onto a new system, automatically installing any dependencies.

3.2.2 ROS Control

ROS Control [10] is a metapackage (`ros_control`) [6], being a set of ROS packages, including controller interfaces, controller managers, transmissions and hardware interfaces. It is essentially a framework that provides the capability to implement and manage robot controllers in the ROS ecosystem.

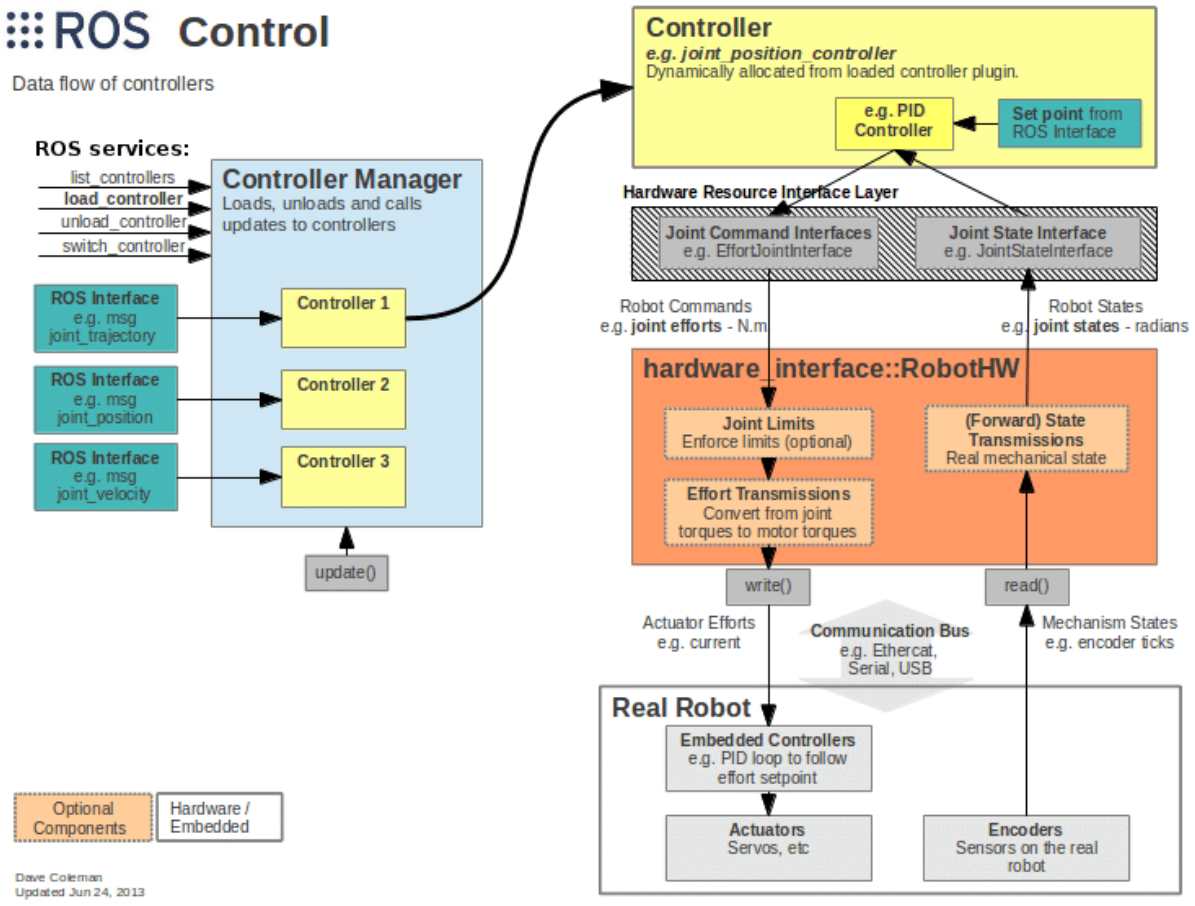


Figure 6: Overview of ROS control components.

The framework consists of several components, as illustrated in Figure 6. Controllers (■) are maintained by the Controller Manager (■). A controller takes as input a setpoint via the ROS Interface (■), which may come from third-party applications, such as Moveit!. The controllers do not directly communicate with the hardware, but do it through the Hardware Abstraction Layer (■, ■), which provides various hardware interfaces (e.g., position, effort etc.), enabling controllers to be hardware agnostic.

The Hardware Abstraction Layer provides an interface (■) such that, e.g., effort commands can be sent to the robot and joint states can be read and used in the control loop. The hardware interface class (■) implements the actual `write()` and `read()` methods that communicate with a specific robot, which must be implemented using one or more of the standard ROS Control interfaces, e.g., `EffortJointInterface` or `VelocityJointInterface`.

The hardware agnostic nature of the framework is what enables a controller to be used across multiple robots or even environments (e.g., physical and simulated), as long as the hardware interface are implemented at the desired endpoint.

Within the hardware interface, the `joint_limits_interface` package allows to (optionally) enforce joint limits on the position, velocity and effort. The limits can be specified in the robot description URDF and override any commands by the controller, before these are actually sent to the hardware.

Furthermore, the hardware interface implements mechanical transmissions, mapping effort/flow variables to output effort/flow variables while preserving power, e.g., a mechanical reducer with some given ratio. These can, like the joint limits, be specified in the URDF file.

3.2.3 Franka ROS

The `franka_ros` [11] metapackage integrates `libfranka` into ROS and ROS Control. It enables to define controllers using the ROS Control framework, providing access to the Panda robot's hardware interfaces using both the standard ROS Control hardware interfaces (position, velocity, effort), but also using the proprietary `franka_hw::FrankaHW` class, which furthermore gives access to interfaces that allow to read the robot state and robot dynamics.

Furthermore, auxiliary packages such as `franka_description` and `franka_visualization` enable further integration into the ROS ecosystem, providing robot description URDF files and the configuration necessary to visualize the robot using RViz.

A controller can be implemented following the standard ROS Control procedure [12]. For the Panda, a controller is implemented by inheriting from the `controller_interface::MultiInterfaceController` class and claiming a valid combination of up to four of the available Franka ROS hardware interfaces, e.g., `EffortJointInterface` + `FrankaCartesianPoseInterface`.

The inherited virtual functions `init()` and `update()` must be implemented as a minimum; any initialization processes must be implemented in the `init()` method whilst the actual control loop is contained in the `update()` method, which is invoked at a frequency of 1 kHz. Furthermore, the `starting()` and `stopping()` methods may also be implemented if necessary.

When implementing a controller, there are several necessary conditions to uphold [13], referred to as joint limits; including limits on joint position, velocity, torque (also referred to as effort) and rotatum (rate of torque). Furthermore, any discontinuities in the trajectory must also be avoided. If the necessary conditions are violated, an error will abort the motion.

3.3 Marker tracking

The structure of marker tracking application is illustrated in Figure 7 and consists of a MOCAP sampling node, a data collection and visualization node using the `plotjuggler` [5] package, and a Cartesian space position controller which interfaces with the robot using ROS Control and Franka ROS.

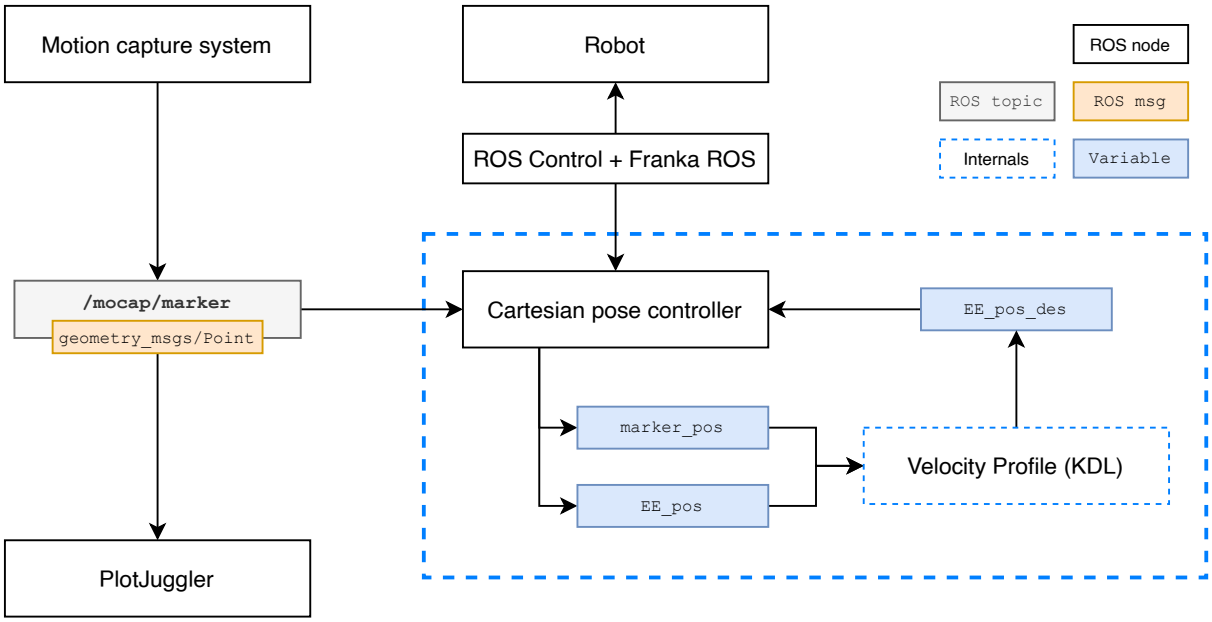


Figure 7: Overview of MOCAP marker tracking application.

The *motion capture system* node communicates with the OptiTrack motion capture system using the `NatNet` SDK [14], allowing to remotely initiate streaming of data and define callbacks to handle it. Three markers are combined into a rigid body in the OptiTrack software (Motive); the position of the rigid body is then streamed, allowing the *motion capture system* node to capture and publish the data to the `/mocap/marker` topic using the `geometry_msgs/Point` [15] message definition.

The *PlotJuggler* node subscribes to the `/mocap/marker` topic and visualizes the position ($[x, y, z]$) of the tracked marker in real time, exemplifying the data collection capability of the HRIC software architecture.

The *Cartesian pose controller* node uses the ROS Control framework and Franka ROS packages to interface with the Panda robot using the `FrankaStateInterface` and `FrankaPoseCartesianInterface` hardware interfaces, allowing to, respectively, read the robot state and command a desired end effector pose using the internal motion generators.

The controller subscribes to the data published by the *motion capture data* node, using it to compute the desired end effector pose (although only position is changed, with orientation of the end effector remaining unchanged). The first time a marker position is received, the initial end effector and initial marker position is set. Any following callbacks are then used to compute the relative difference in the position of the marker, which are in turn used to to compute the desired absolute end effector position.

The marker and end effector do not share a common coordinate system (frame of reference); the marker position, ${}^{M}\boldsymbol{p}$ is given in the frame of reference of the MOCAP system, $\{M\}$, whereas the end effector position is given in the robot base frame, $\{B\}$. The marker position is desired in the robot base frame, i.e., ${}^{B}\boldsymbol{p}$, which is given by the transformation

$$ {}^{B}\boldsymbol{p} = {}^{B}_{M}\mathbf{R}\,{}^{M}\boldsymbol{p} \qquad \textit{where} \qquad {}^{B}_{M}\mathbf{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}. $$

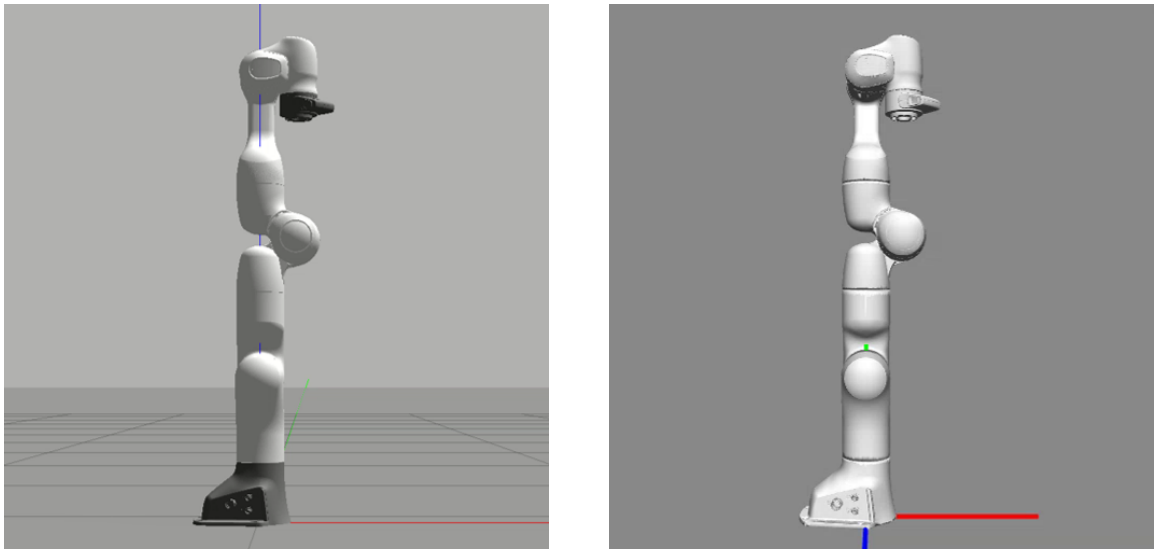
In order to comply with the Panda's necessary conditions, specifically the Cartesian velocity and acceleration limits, the desired end effector position cannot be commanded directly, and an interpolated trajectory must instead be computed with some specified acceleration and velocity profile, which is implemented using KDL, specifically using the `KDL::VelocityProfile_Trap` class.

The actual control loop consists of reading the current robot state and appending the desired end effector position to the current robot pose, where the desired end effector position is read from the trapezoidal velocity profile (trajectory) as a function of time. Every time a new marker position is received, the controller attempts to compute a new trajectory, unless a trajectory is already in execution; this is to prevent discontinuities in the joint space. This approach greatly limits the responsiveness of the end effector to the marker motion, but is sufficient as a proof of concept.

3.4 Evaluation

The Cartesian pose controller is launched using the `roslaunch` system, connecting to the Panda robot and automatically subscribing to the (yet not advertised) marker position topic. Using the OptiTrack Motive software, the position of the set of markers which constitute the rigid body are calibrated to align with the axes of the MOCAP system, after which the *motion capture system* node can be instantiated using `rosrun`, advertising the position of the marker.

Finally, the *PlotJuggler* node is also instantiated using `rosrun` and configured to read data from the MOCAP sampler node's `/mocap/marker` topic, facilitating a real time visualization of the data, as shown in Figure 8b.



(a) Snapshots of Panda end effector movement.

(b) MOCAP data visualized in PlotJuggler.

Figure 8: PLACEHOLDER FIGURES!

As evidenced by the videographic demonstration in [16], the end effector of the robot is able to successfully track the motion of marker, mimicking the relative changes in position, as illustrated in Figure 8a.

The modularity of the architecture is demonstrated by the possibility of implementing the processes of the marker tracking system independently, furthermore enabling the use of third party packages such as the `plotjuggler` package.

Since the FCI requires real time control, the ROS framework is indeed successful in enabling real time interfacing of the robot whilsts also providing a communication framework for data collection.

Chapter 4

Framework design for generic controllers

4.1 Overview

A generic controller is an arbitrary controller which exhibits similar behavior in both reality and simulation. For the HRIC platform, the development of a framework that enables such controllers requires a simulation environment to be configured.

ROS and Gazebo can be used to facilitate an integrated simulation environment which makes use of a robot model with estimated dynamic parameters to simulate a robot, providing an emulated hardware interface that allows to control the robot and access both the robot state and the robot dynamics.

4.2 Simulation environment

Gazebo [17] is an open-source 3D robotics simulator and is one of primary tools used in the ROS community, supporting different physics engines (ODE, Bullet, SimBody and DART). It can be used in addition with ROS to simulate robots that are interfaced using the ROS Control framework.

4.2.1 Integration of robot model

In the Gazebo simulation environment, URDF can be used to describe a robot model, herein the kinematic, visual, collision and dynamic properties of a robot. The Panda provides a ROS package with the robot description [18], which includes URDF files and meshes. These define only the kinematic, visual and collision properties of the robot and do not contain the dynamic parameters, which are required for dynamic simulations in Gazebo. It is therefore necessary to define a model in which both the kinematic and dynamic properties are defined.

The process of integrating the Panda into a Gazebo simulation environment consists of:

1. Fixing the robot to the world coordinate system
2. Adding damping to the joint specifications
3. Adding dynamic parameters (inertia, mass and friction)

The robot description URDF file is divided into several parts and constructed using the Xacro (XML Macros) [19] ROS package; Xacro is an XML macro language which allows to construct shorter and more readable XML files by using macros that expand to larger XML expressions. It provides support for arguments and expressions, allowing to include other files and more.

The robot is constructed in the `panda.urdf.xacro` parent file, as shown in Listing 1, which can be converted to a pure URDF file using the `xacro` command-line tools. In this file, the `panda_arm.xacro` and `panda.gazebo.xacro` files are included.

```
<?xml version="1.0" encoding="utf-8"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="panda">

  <!-- robot name -->
  <xacro:arg name="robot_name" default="panda" />

  <!-- load arm + hand -->
  <xacro:include filename="$(find franka_description)/robots/panda_arm.xacro" />

  <!-- load gazebo files -->
  <xacro:include filename="$(find franka_description)/robots/panda.gazebo.xacro" />

  <!-- spawn robot -->
  <xacro:panda_arm />
  <xacro:panda_gazebo robot_name="$(arg robot_name)" />

</robot>
```

Listing 1: Simplified example of robot description using Xacro (`panda.urdf.xacro`).

The `panda_arm.xacro` file contains definition of the Panda robot arm, using `<link>` and `<joint>` tags to define the kinematic chain. In the `<joint>` tag, the nested `<dynamics damping="1.0">` tag can be used to specify the joint damping, whereas the `<link>` tag has the nested `<visual>`, `<collision>` and `<inertial>` tags to respectively specify the visual, collision and inertial properties.

The `panda_arm.xacro` file contains any Gazebo specific properties for e.g. links and joints [20], with tags used to define material information, whether the joints provide feedback etc.

Extending the provided `franka_description` URDF file, the robot can be fixed to the world coordinate system by adding a world link and a fixed joint to connect the world frame to the base link of the robot, as shown in Listing 2.

```
<link name="world" />

<joint name="robot_to_world" type="fixed">
  <parent link="world" />
  <child link="${arm_id}_link0" />
  <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
  <dynamics damping="${joint_damping}"/>
</joint>
```

Listing 2: Simplified example of robot description using Xacro (`panda.urdf.xacro`).

For the revolute joints of the Panda robot, the joint damping property is added using the aforementioned `<dynamics>` tag, which is added to each `<joint>` tag, similar to that of the `robot_to_world` joint in Listing 2. The joint damping can be made variable using Xacro, such that the same parameter is applied to all the joints.

Since Franka does not provide the actual dynamic parameters, these must be estimated as proposed by [21], where the used parameters can be found at [22]. However, the inertia matrix of the estimated parameters is defined with respect to the link origin frame, where Gazebo needs the inertia matrix to be defined with respect to the link inertia frame (as shown in Figure 3b), which is located at the center of gravity.

The estimated inertia matrices can be transformed to the CoM using Steiner's theorem [23, p. 287], which states that the inertia matrix $\boldsymbol{\mathcal{I}}_q$ described with respect to a frame (denoted by curly braces) aligned with $\{b\}$, but at point $\boldsymbol{q} = [q_x, q_y, q_z]^T$ in $\{b\}$, is related to the inertia matrix $\boldsymbol{\mathcal{I}}_b$ calculated at the center of mass, by

$$\boldsymbol{\mathcal{I}}_q = \boldsymbol{\mathcal{I}}_b + m(\boldsymbol{q}^T\boldsymbol{q}\mathbf{I} - \boldsymbol{q}\boldsymbol{q}^T), \tag{1}$$

where $\mathbf{I}$ is the 3×3 identity matrix and m is the mass of the rigid body.

Thus, given the inertia matrix at the link origin, $\boldsymbol{\mathcal{I}}_q$, it is related to the inertia matrix, $\boldsymbol{\mathcal{I}}_b$ (defined at the center of gravity), by the translation, $\boldsymbol{q}$. It is possible to compute $\boldsymbol{\mathcal{I}}_b$ as

$$\boldsymbol{\mathcal{I}}_b = \boldsymbol{\mathcal{I}}_q - m(\boldsymbol{q}^T\boldsymbol{q}\mathbf{I} - \boldsymbol{q}\boldsymbol{q}^T), \tag{2}$$

Looking at an example for the first link, with the estimated inertia matrix, $\boldsymbol{\mathcal{I}}_q$ [kg/m^2], center of mass, $\boldsymbol{q}$ [m], and mass, m [kg], given as

$$\boldsymbol{\mathcal{I}}_q = \begin{pmatrix} 0.7034 & 0.0001 & -0.0068 \\ 0.0001 & 0.7066 & -0.0192 \\ -0.0068 & -0.0192 & 0.0091 \end{pmatrix}, \qquad \boldsymbol{q} = -\begin{pmatrix} 0.0039 \\ 0.0021 \\ -0.1750 \end{pmatrix}, \qquad m = 4.9707,$$

the transformed inertia of matrix, $\boldsymbol{\mathcal{I}}_b$, can be computed using Equation 2 to

$$\boldsymbol{\mathcal{I}}_b = \begin{pmatrix} 0.5511 & -0.0001 & 0.0034 \\ -0.0001 & 0.5543 & 0.0174 \\ 0.0034 & 0.0174 & 0.0090 \end{pmatrix}.$$

The resulting inertia matrices do not always satisfy triangle inequality, meaning that such would not be feasible in the physical world. This is presumably due to the estimated nature of the parameters, only being an approximation.

4.2.2 Gazebo and ROS Control

The ROS control (`ros_control`) framework can be integrated with Gazebo [7], allowing to use the framework in order to implement a controller and thus actuate the joints of the simulated robot by accessing the hardware interfaces of ROS Control (position, velocity and effort) - the dataflow is illustrated in Figure 9.

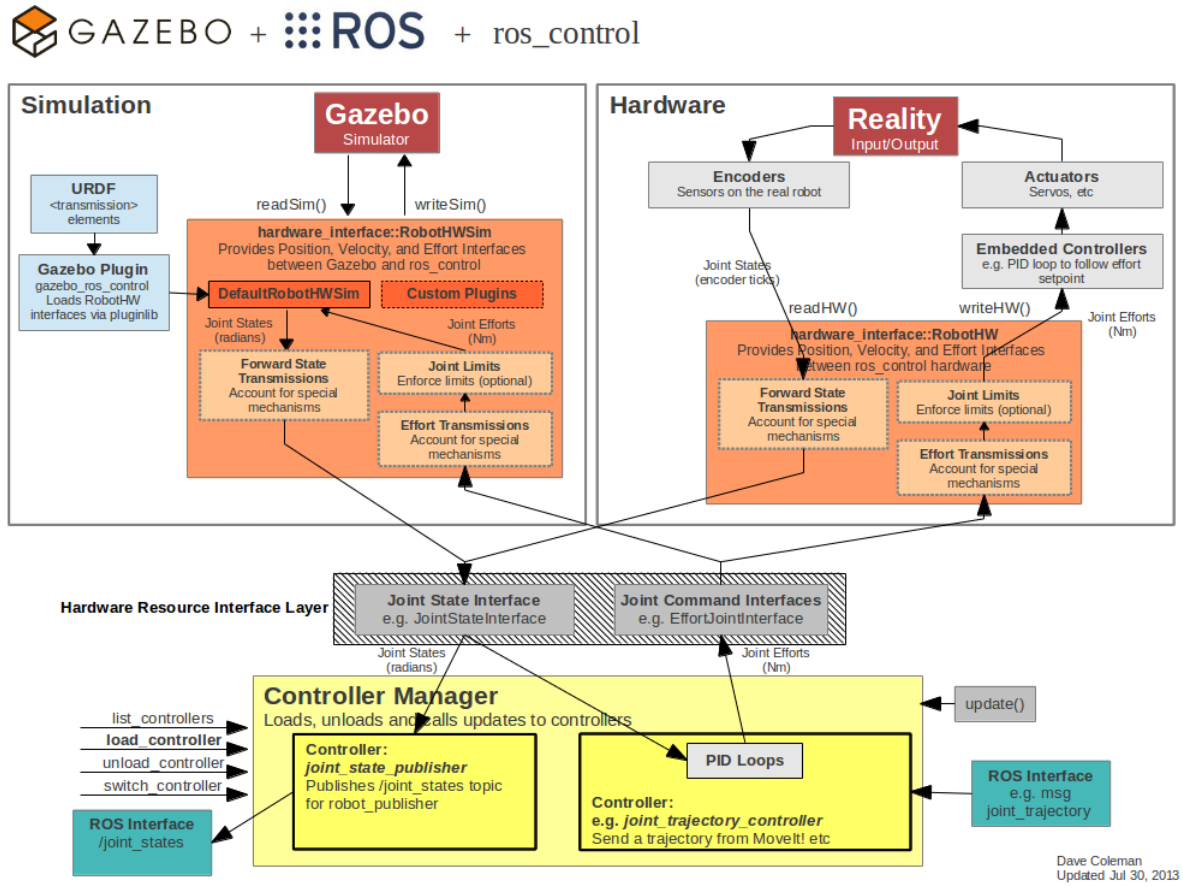


Figure 9: Relationship between simulation, hardware, controllers and transmissions of ROS Control and Gazebo.

This integration is enabled by the `gazebo_ros_pkgs` set of packages [24], in which a Gazebo plugin adapter uses the abstract classes of `ros_control` to define a `hardware_interface::RobotHWSim` class with `readSim()` and `writeSim()` methods, establishing an interface between Gazebo and `ros_control`.

The hardware agnostic structure of the ROS Control framework, specifically the interface (■) provided by the Hardware Abstraction Layer, is what allows to seamlessly integrate ROS Control with Gazebo, providing a framework for development of generic controllers, as long the standard ROS Control hardware interfaces are used.

In order to activate a controller in the simulation, a transmission file (`panda.transmission.xacro`) is defined and included in the parent file (`panda.urdf.xacro`). The transmission file contains information necessary for Gazebo to simulate the control behavior; the `<transmission>` tag is used to link actuators to joints, specifically defining the transmission type and hardware interface for each joint together with the actuator itself, as shown in Listing 3.

```
<transmission name="${robot_name}_tran_1">

    <type>transmission_interface/SimpleTransmission</type>

    <joint name="${robot_name}_joint1">
      <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
    </joint>

    <actuator name="${robot_name}_motor_1">
      <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>

</transmission>
```

Listing 3: Transmission example for joint1 which links link0 and link1.

The `the gazebo_ros_control` Gazebo plugin parses the transmission tags and loads the appropriate hardware interfaces and controller manager; this plugin is loaded as shown in Listing 4.

```
<gazebo>
   <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
   </plugin>
</gazebo>
```

Listing 4: Adding of Gazebo plugin into URDF.

The actuators are defined using the `EffortJointInterface` hardware interface, allowing to actuate the joints using an effort controller, e.g., `effort_controllers/JointTrajectoryController`.

The `franka_description` package is modified with the integrated URDF model. A package named `franka_gazebo` is created to bundle all necessary functionality regarding a simulated environment, specifically a `roslaunch` launch file (`launch/sim.launch`) and a configuration file (`config/panda_control.yaml`). The launch file automates the process of loading the integrated system of Gazebo and ROS Control, where it:

1. Locates and loads the robot description (URDF)
2. Launches a Gazebo environment (e.g., empty world)
3. Spawns the robot in Gazebo
4. Locates and loads ROS controller specifications (e.g., PID gains)
5. Loads the `ros_control` controller manager and spawns the controllers

The configuration file specifies controller configurations, where any defined controllers can then be spawned via the Controller Manager in the launch file. The used controllers must match with the hardware interfaces specified in the transmission file.

For example, the `effort_controllers/JointTrajectoryController` [25] controller is configured in the `config/panda_control.yaml` file, specifying the robot's joints, PID gains and so forth, as exemplified in Listing 5.

```
panda_joint_trajectory_controller:

    type: effort_controllers/JointTrajectoryController
    state_publish_rate: 100

    joints:
        - panda_joint1
        - panda_joint2
        - ...

    gains:
        panda_joint1: { p: 12000, d: 50,  i: 0.0,  i_clamp: 10000 }
        panda_joint2: { p: 30000, d: 100, i: 0.02, i_clamp: 10000 }
        ...

    constraints:
        goal_time: 2.0
```

Listing 5: Configuration of `joint_trajectory_controller` in the `panda_config.yaml` file.

The launch file is then used to initialize the system and load the robot into the simulation environment, as illustrated in Figure 10.

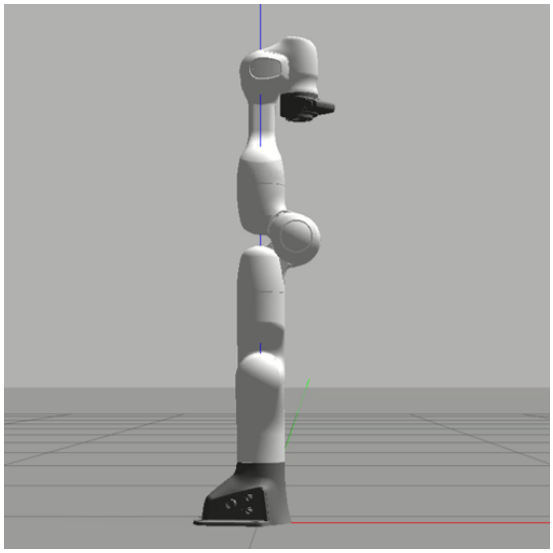


Figure 10: Simulated Panda robot in Gazebo.

The `panda_joint_trajectory_controller` controller is spawned as specified in the configuration, exposing a ROS interface for specifying a joint trajectory, meaning that a desired joint trajectory can be defined using the `trajectory_msgs/JointTrajectory` message and issuing a command to the controller's command topic, in this case `panda_joint_trajectory_controller/command`.

4.3 Generic PD controller

Once the integrated simulation environment is initialized, ROS Control can be used to implement custom controllers using the standard hardware interfaces or by defining custom hardware interfaces.

A joint space PD control with gravity compensation [26], as shown in Figure 11, is able to satisfy the position control objective globally for n-DoF robots; it requires both the robot joint states and online computation of the gravity term $\boldsymbol{g}(\boldsymbol{q})$, making it suitable for a simple test of the generic controller framework.

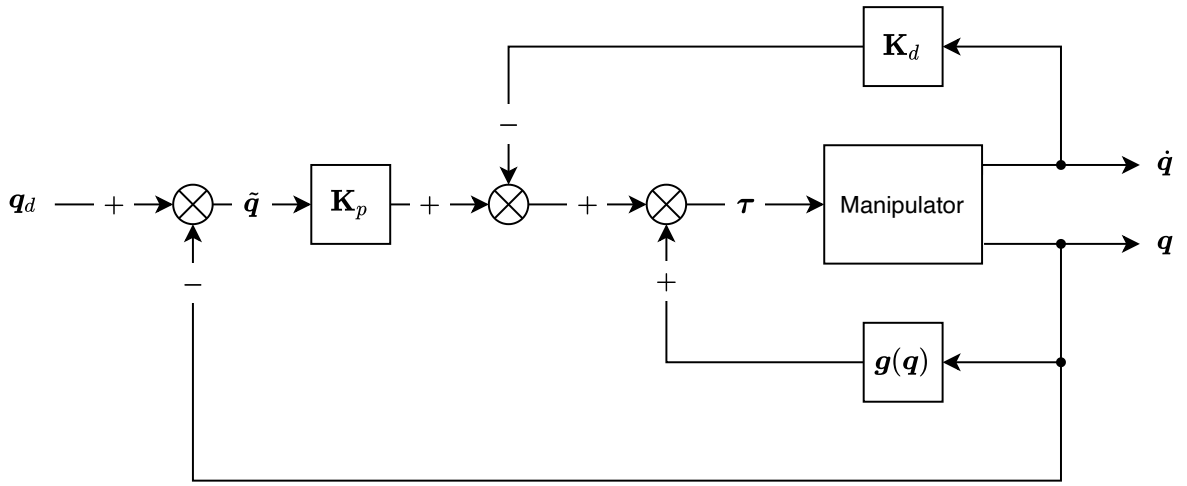


Figure 11: Joint space PD control with gravity compensation.

The controller input is a vector of desired joint positions, $\boldsymbol{q}_d \in \mathbb{R}^n$, with the vectors $[\boldsymbol{q}, \dot{\boldsymbol{q}}]$ defining the system state, where

$$\tilde{\boldsymbol{q}} = \boldsymbol{q}_d - \boldsymbol{q}$$

represents the error between the desired and the actual posture. The PD control law with gravity compensation, specifying the desired torque vector, $\boldsymbol{\tau}$, is given by

$$\boldsymbol{\tau} = \mathbf{K}_p \cdot \tilde{\boldsymbol{q}} - \mathbf{K}_d \cdot \dot{\boldsymbol{q}} + \boldsymbol{g}(\boldsymbol{q}) \tag{3}$$

where $\mathbf{K}_p, \mathbf{K}_d \in \mathbb{R}^{n \times n}$ are symmetric positive definite matrices.

4.3.1 Hardware interface emulation

In order to implement the joint space PD controller with gravity compensation in Gazebo, the gravity term, $\boldsymbol{g}(\boldsymbol{q})$, must be available at each instance of the control loop. The robot state $[\boldsymbol{q}, \dot{\boldsymbol{q}}]$ can be extracted from the `JointStateInterface` hardware interface, but the robot dynamics, i.e., the mass matrix, $\mathbf{M}(\boldsymbol{q})$, Coriolis matrix, $\mathbf{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$, and gravity vector, $\boldsymbol{g}(\boldsymbol{q})$ in the joint space dynamic model of the Panda in (4), which are typically accessed via `libfranka`'s hardware interface, are not available in Gazebo and must somehow be emulated.

Computing the position dependent gravity term, $\boldsymbol{g}(\boldsymbol{q})$, is an degenerate inverse dynamics problem, i.e. the problem of computing joint forces and torques corresponding to the robot state, as described by the general equation of motion

$$\boldsymbol{\tau} = \mathbf{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \mathbf{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{g}(\boldsymbol{q}). \tag{4}$$

Dynamic controllers, such as the PD gravity compensation controller or an impedance controller, rely on the availability of the robot dynamics, where such properties can be extraordinarily complex analytically, even for simpler systems [23, p. 271].

Given the kinematic state of the robot, a rigid-body dynamics library is able to compute the robot dynamics. Orocos Kinematics and Dynamics Library (KDL) [27], which is available as a ROS package, provides the dynamic solver class `KDL::ChainDynParam`. It uses the kinematic chain, which can be loaded from the URDF robot description, to compute the robot dynamics for a given robot state.

The functionality of the KDL solver is wrapped into a utility class and added to the `franka_gazebo` package, which can be included as `<franka_gazebo/dynamics.h>`. It subscribes to the robot states published by `joint_state_controller` and computes the dynamic properties every time the callback is invoked, providing an emulated hardware interface. It also allows to manually compute the robot dynamics with a function call, given the robot state.

4.3.2 Controller implementation

The simulation controller is derived from the `controller_interface::Controller` class using the `EffortJointInterface` hardware interface for joint actuation, which also exposes the `JointStateInterface` hardware interface for reading the joint states.

The control loop is implemented by overriding the virtual `update()` method, as shown in Listing 6. First, the desired joint positions are read from the ROS interface; here the `commands_buffer` container from the `realtime_tools` package [28] is used to bridge the real time control loop and non-real time ROS interface.

The robot state $[\boldsymbol{q}, \dot{\boldsymbol{q}}]$ is acquired using the `JointStateInterface` hardware interface. By including `<franka_gazebo/dynamics.h>` from the `franka_gazebo` package, the robot state can be used to directly compute the emulated robot dynamics, leveraging the necessary gravity vector, $\boldsymbol{g}(\boldsymbol{q})$.

Using the desired joint positions, the robot state and the emulated robot dynamics, the control law, as given by Equation 3, can be implemented, computing the desired torque.

To ensure that the specified joint limits of the Panda are complied with, the rotatum of the desired torque is saturated. Position, velocity and torque joint limits are automatically enforced by the hardware interface class, since these were defined in the URDF robot description.

The saturated desired torque is then commanded to the robot's actuators using the `EffortJointInterface` hardware interface through a vector of joint handles.

```
void
update(const ros::Time& time, const ros::Duration& period) override
{
    // PID gains
    constexpr double kp = 50.0, kd = 10.0;

    // get desired joint positions (from non-RT thread)
    std::vector<double>& q_d = *commands_buffer.readFromRT();

    // read joint states
    const auto q    = get_position(vec_joints);
    const auto qdot = get_velocity(vec_joints);

    // compute dynamics
    const auto g = franka_gazebo::dynamics::gravity(q);

    // compute controller effort
    const auto tau_des = kp * (q_d - q) - kd * qdot + g;

    // encorce rate of torque (rotatum) saturation
    const auto tau_des_sat = saturate_rotatum(tau_des);

    // set desired command on joint handles
    for (size_t i = 0; i < num_joints; ++i)
        vec_joints[i].setCommand(tau_des_sat[i]);
}
```

Listing 6: Control loop of simulated PD gravity compensation `ros_control` controller.

The implementation of the controller on the real robot is similar to that of the simulated controller, although some modifications are necessary. Like in the simulated controller, the `EffortJointInterface` is used to command joint torques, also exposing the `JointStateInterface` to directly acquire the robot state. However, the the robot dynamics need not be emulated and can be acquired directly using the proprietary `FrankaModelInterface` hardware interface.

Since the Panda's internal controllers provide both gravity and friction compensation for any torque commands via the `JointEffortInterface`, adding the gravity term to the control law is not necessary.

4.4 Evaluation

The behavior of the real and simulated controllers is compared by starting the robot in roughly the same joint configurations with all joint positions set to zero. The controllers are then commanded a setpoint of $\boldsymbol{q}_d = \left[0, -\frac{\pi}{4}, 0, -\frac{3\pi}{4}, 0, \frac{\pi}{2}, \frac{\pi}{4}\right]$, resulting in a motion as seen in Figure 12. The PD values are empirically chosen to $K_p = 200$ and $K_d = 100$ for the simulated robot and $K_p = 10$ and $K_d = 5$ for the real robot.

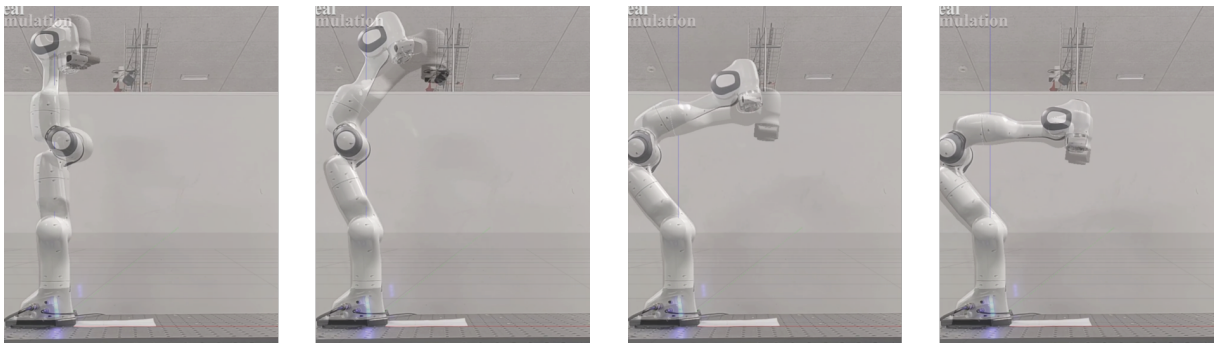
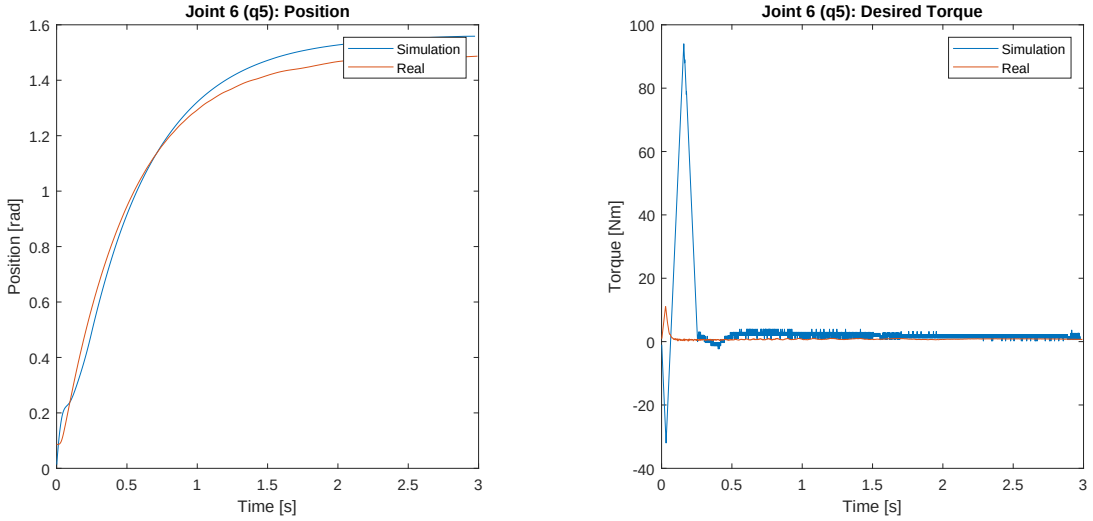


Figure 12: Overlayed snapshots of the simulated and real robot moving.

During the execution, various parameters, such as joint position, desired torque, the gravity term and so forth are logged in the control loop and written to a file. The movement of the robot is recorded and comapred to the simulation in [29].



(a) Joint position of of sixth joint.

(b) Commanded desired torque of sixth joint.

Figure 13: Joint states of simulated and real robot during execution.

Looking at the logged position of the sixth joint (q_5), as shown in Figure 13a, the trajectories are similar with a difference between the simulated and real joint position being seemingly small, peaking at a difference of 0.0886 rad at 0.2260 s and steady state error of 0.0720 rad at ~ 3 s.

The desired joint torque of the sixth joint (τ_{d_5}), as shown in Figure 13b, exhibits a greater difference during the initial joint movement. However, comparing the desired torque of the simulated and real robot is inherently erroneous, since the FCI internally compensates the desired joint torque for both gravity and friction, resulting in a different commanded torque.

In general, joint positions in simulation deviate somewhat from that of the real robot but exhibit a similar trajectory with some steady state errors, as evidenced in Figure 14 and in Figure 12. Optimizing the PD values would presumably better the simulated behavior, although ideally, emulating the friction and having access to the more accurate dynamic parameters is desirable.

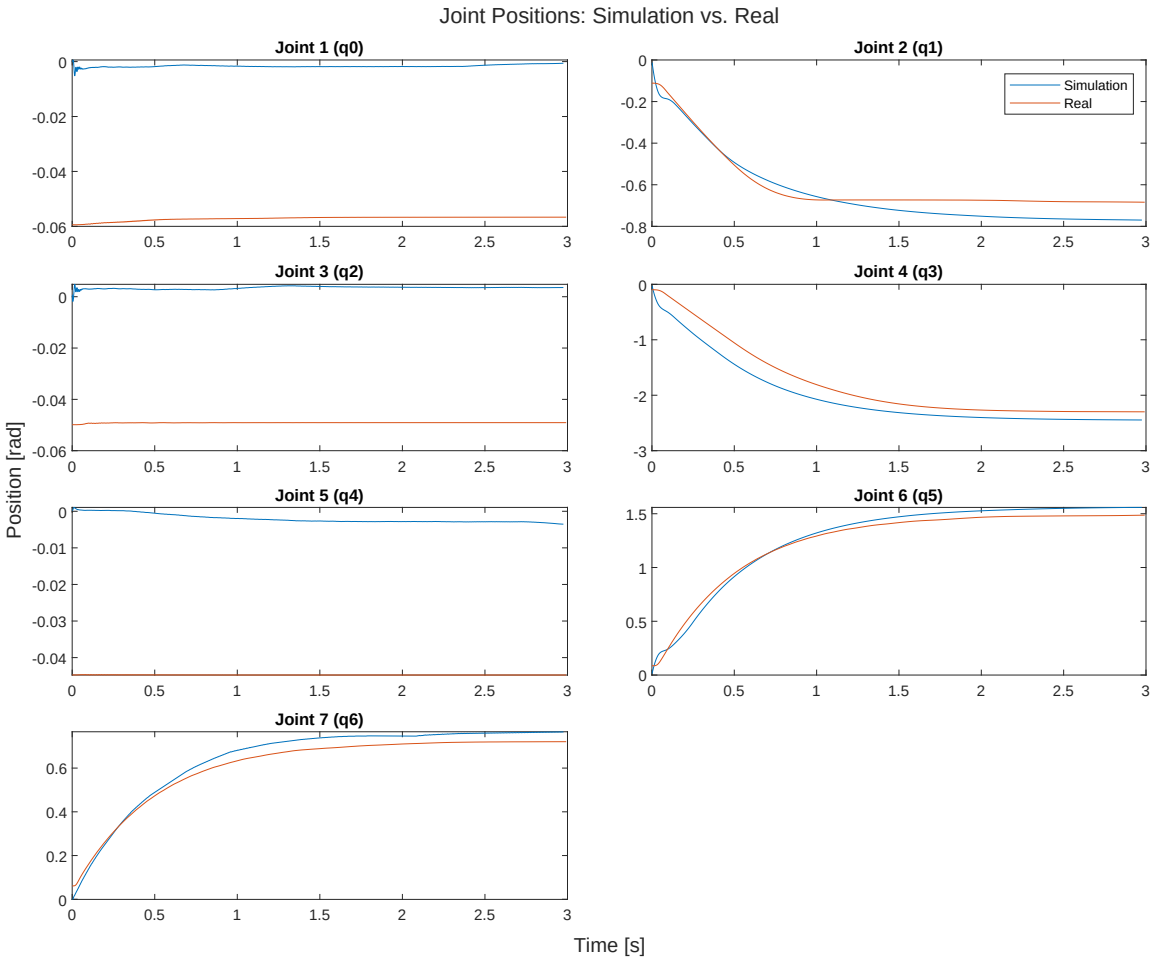


Figure 14: Joint positions of the simulated and real robot.

The current controller framework solution is somewhat generic, but does not provide a fully generic framework for controller design, since the control loops are not identical in terms of their implemented control laws. However, given the correct dynamic parameters of the Panda, a fully generic sim-to-real controller framework is theoretically possible using the ROS Control and Gazebo ROS framework.

One possible approach consists of a two parts: (a) a custom robot hardware interfaces between ROS Control and Gazebo, and (b) a generic hardware interface for the robot dynamics.

The Gazebo `gazebo_ros_control` plugin can be used to implement a custom simulated hardware interface by inheriting the `gazebo_ros_control::RobotHWSim` class, allowing to internally emulate the robot dynamics (e.g. using KDL) and compensate the desired joint torque for both gravity and friction. Once modified, the inherited class can be specified to load in the aforementioned transmission file (`panda.transmission.xacro`), as shown in Listing 4, using the `<robotSimType>` tag. This would allow to implement the same control law in both the simulated and real control loop, since both desired joint torques would be automatically (internally) compensated.

In order to provide a hardware interface that gives access to the robot dynamics, a generic model hardware interface could be inherited from the `franka_hw::FrankaModelInterface` class, which would provide the robot dynamics in the same format, albeit automatically detect if the robot is being simulated and emulate the robot dynamics when necessary. In combination with the automatically compensated desired joint torques, this would allow to define the control loop of a dynamics controller as a single component, with access to the robot dynamics via a shared interface, to be used on both the simulated and real robot.

The only remaining issue is with regard to launching of the system, yet such a process is trivial to automate using `roslaunch`, where the same controller could be loaded into either a simulated or real environment by simply providing an argument, e.g., `env = "sim" || "real"`.

Chapter 5

Robot simulation in biomechanical simulator

5.1 Overview

In order to analyze a robot with an arbitrary controller in contact with a musculoskeletal model of the human body, a simultaneous simulation of the human and robot is required, i.e., simultaneously being able to model and simulate both (a) the rigid body kinematics and dynamics of a robot and (b) the musculoskeletal kinematics and dynamics of a human.

One possibility is to integrate the simpler, robotic model into a biomechanical simulator. OpenSim is open source software for biomechanical modeling, simulation and analysis. It provides widely accessible tools for conducting biomechanics research and motor control science, being one of the most widespread biomechanical simulators.

The URDF robot descriptions must be transformed to OpenSim proprietary XML based modeling markup language (.osim). Once a model is loaded, it is of interest to research whether the controllers implemented in ROS control exhibit a familiar behavior in the OpenSim biomechanical simulator.

5.2 Model transformation

An OpenSim model [30] is described by an XML-like file format and allows to represent the neuromuscular and musculoskeletal dynamics of (typically) a human, which is to be analyzed via simulation. The model is comprised of various components, as illustrated in Figure 15, consisting of reference frames, bodies, joints, constraints, forces, contact geometry, markers and controllers.

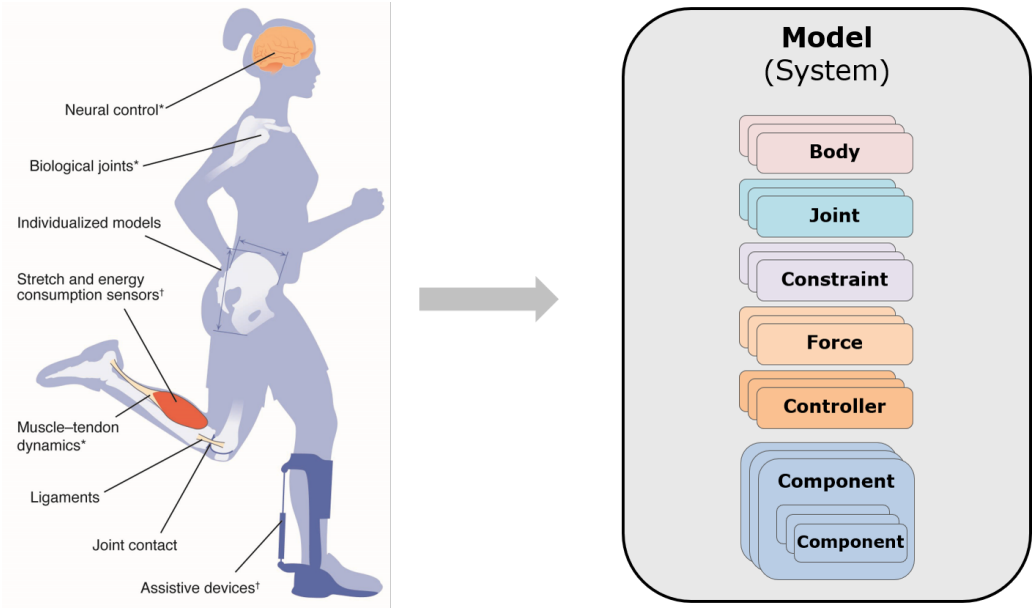


Figure 15: Components of an OpenSim model.

In order to transform the URDF robot model into the OpenSim Model Format (OSIM), the links of the URDF model must be defined as OpenSim bodies, including dynamic properties, and then be connected via OpenSim joints; there are several types of joints in OpenSim, which can be used to model the regular robot joints; fixed robot joints can be modeled as the WeldJoint type, prismatic robot joints can be modeled as the SliderJoint type, whereas revolute joints can be modeled as the PinJoint type, albeit there is no idiomatic way to add joint friction and damping. Instead, a force component can be used to model any forces due to contact, such as surface friction.

Manually defining the OpenSim model can be quite cumbersome; instead, using the OpenSim `C++` library, a model can be programmatically constructed and then exported as an OSIM file. Furthermore, a URDF parser can be used to load the robot description and potentially be used to automate the model transformation process, at least to some degree; such functionality is implemented and bundled as the `urdf_to_osim` package.

An automated transformation process consists of first parsing the URDF robot description and then iterating the links of the loaded model (excluding the world and the last link), from which the OpenSim model is sequentially constructed. For each link, various properties are extracted from the URDF model, including the link number, link dynamics (mass, inertia, CoM) and so forth.

To connect a parent and child link with a joint, it is necessary to define the position and orientation of the joint in it's parent link with respect to the child link. For each iterated (child) link, information is extracted about the parent joint, containing the necessary information in the parsed URDF model. Each joint is then defined as a PinJoint and connected to the previously iterated link. The first link, being a special case, is connected to the ground frame using the WeldJoint type, whereas the last link is ignored due to the URDF model being defined following the modified DH parameters convention.

In order to make the robot controllable in OpenSim, actuators can be added to joints, which apply pure forces or torques that are directly proportional to the input control; the CoordinateActuator type actuator allows to specify generalized forces that are applied along the axis of a generalized coordinate (i.e., a joint axis). These are added for all the revolute joints of the iterated links.

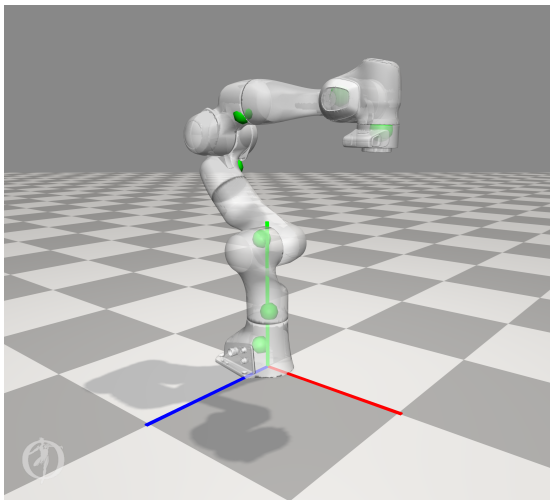


Figure 16: Panda robot in OpenSim with center of mass (CoM) of links shown.

The visual and contact geometry is attached to the rigid bodies by specifying their respective mesh files, resulting in a complete integration of the robot model as seen in Figure 16.

5.3 Controller implementation

The specified joint actuators of the OpenSim model can be controlled by implementing a controller plugin, derived from the `OpenSim::Controller` class, which is then added to the model. In order to load the plugin into the OpenSim GUI, it must first be compiled as a dynamic-link library (DLL) (`.dll` file) and added to the OpenSim plugin directory.

Within the controller class, the virtual method `computeControls()` is overridden to implement the intended control law, e.g., the PD gravity compensation controller, having access to the current state of the system and a vector of controls. The robot state can be acquired from the state variable, but there is no native support for acquiring the robot dynamics, which must therefore emulated like in the case of Gazebo.

The emulation of the simulated robot dynamics cannot be done using KDL, since the controller plugin is compiled as a DLL (to be used with the GUI) outside of ROS. Instead, an algorithm as given by [21] can be used.

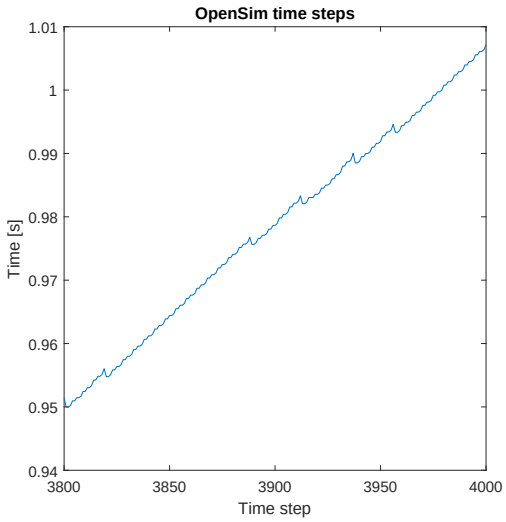


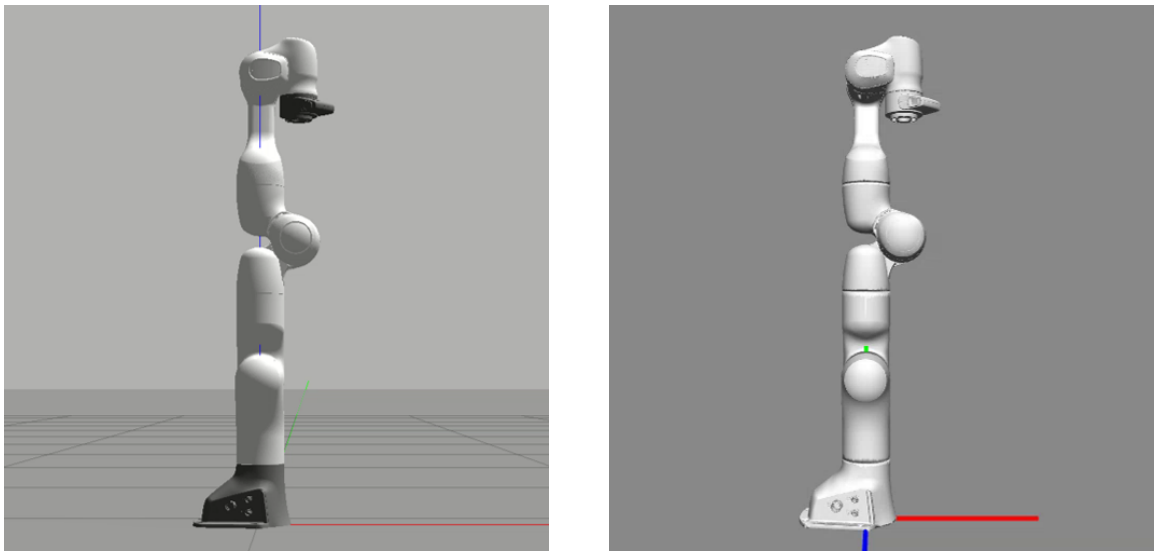
Figure 17: Logged time steps from the OpenSim PD controller control loop.

Another issue with the controller implementation is with regard to limiting the rotatum. The OpenSim simulation integrator functions in such a way that the time is not unidirectional, as evidenced by logging the time steps provided by the state, illustrated in Figure 17. Furthermore, the lack of support for joint damping may also cause the OpenSim simulation to exhibit different behavior compared to the Gazebo simulation.

5.4 Evaluation

The behavior of the Panda robot in OpenSim and its associated controller is compared to its counterpart in Gazebo, similar to the comparison between the simulated and real robot in Section 4.4. To equalize the comparison, the Gazebo robot model in the Gazebo simulation environment has been been altered, removing joint friction and joint limits, since these aren't implemented in the OpenSim robot model.

Furthermore, the Gazebo controller is altered to use the same algorithm as OpenSim for emulating the robot dynamics instead of KDL. The saturation of rotatum is disabled, since that too is not implemented in the OpenSim controller.



(a) Gazebo (b) OpenSim

Figure 18: Initial poses of the Panda robot in its respective simulation environments.

The robot models have their joint positions set to zero, as shown in Figure 18, and are the controllers are then commanded a setpoint of $\boldsymbol{q}_d = \left[0, -\frac{\pi}{4}, 0, -\frac{3\pi}{4}, 0, \frac{\pi}{2}, \frac{\pi}{4}\right]$. Both controllers use the PD values of $K_p = 50$ and $K_d = 10$.

The data is collected in the control loop, using the same methodology as in Section 4.4. Although, in order to compare the OpenSim data to the Gazebo data, the OpenSim data must be filtered due to the non-unidirectional time steps as a result of the integrator's trial computations.

The robots are visually observed to exhibit a similar behavior, as evidenced in [31], where the trajectories of the joint positions are seemingly equal, as shown in Figure 19, settling at similar values after $\sim 3\,\mathrm{s}$, although the model in Gazebo is observed to oscillate more.

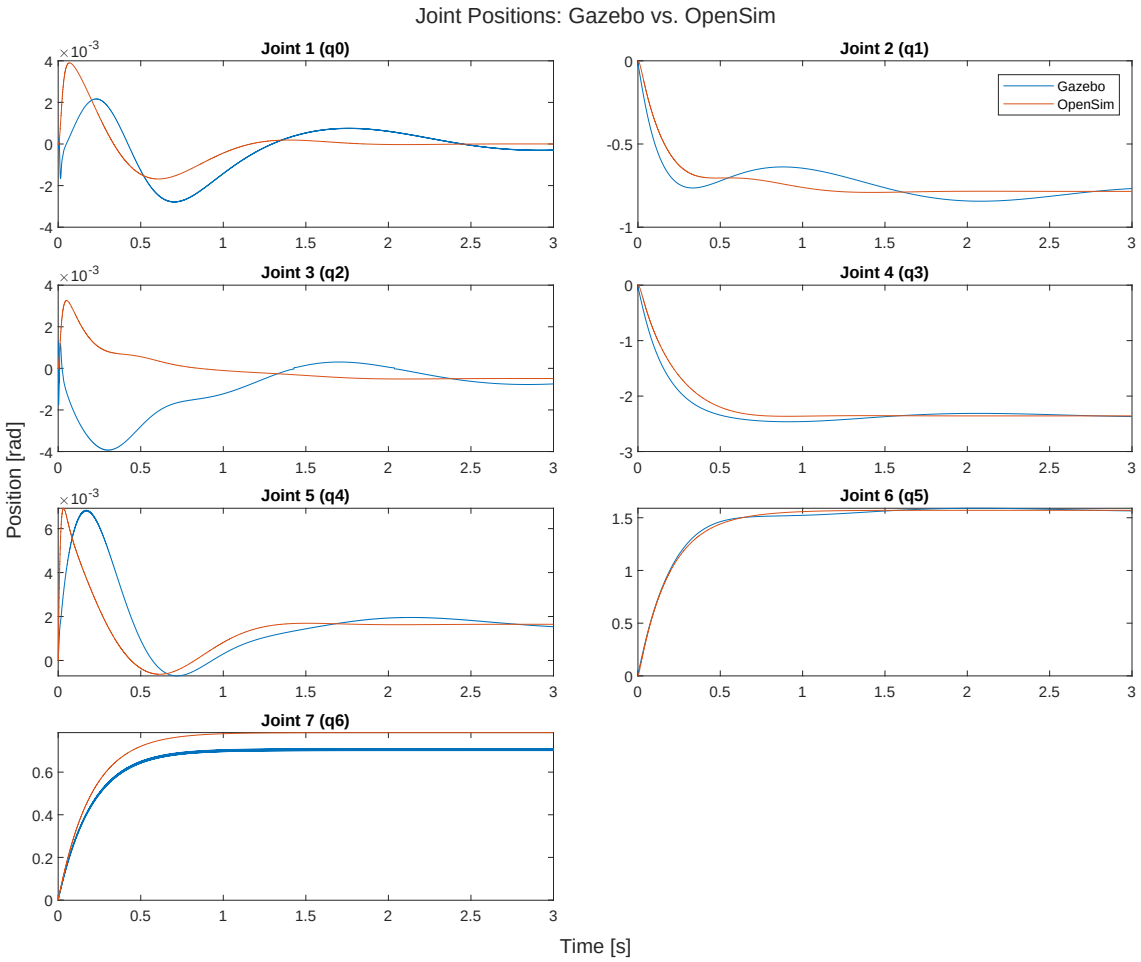


Figure 19: Joint positions of the simulated robot in Gazebo and OpenSim.

Despite the similar behavior of the different simulation environments, OpenSim still lacks native support for joint limits, joint damping and friction. Furthermore, the trial computations of the OpenSim integrator makes it difficult to implement error based control, although presumably possible using OpenSim's state management, as discussed in [32]. Even with state management, the process of implementing a controller in OpenSim is rather cumbersome (compared to ROS Control). The lack of an out-of-the-box GUI solution for Linux further diminishes the case of integrating a URDF robot model in OpenSim, since Franka ROS does not support Windows, which is why using Linux is essential for a generic platform.

If joint limits, joint damping and joint friction can be added to the OpenSim model, assuming these would exhibit a similar behavior to Gazebo, it would be possible to use OpenSim as a standalone platform for biomechanical analysis of Human-Robot Interaction and Collaboration, still using the ROS framework for data collection, albeit without a generic controller framework; at best, a scripting language can be used to abstract and automate the implementation of controllers in ROS Control and OpenSim.

An alternative approach could be one similar to that of the Neurorobotics Platform (NRP) of the Human Brain Project (HBP), having forked Gazebo and implemented a plugin which provides OpenSim as an additional physics engine alongside the physics engines already supported by Gazebo [33].

Their plugin supports many of SimBody's kinematic constraint types and implements collision detection support for sphere, plane and triangle mesh shapes along with corresponding contact forces (as exposed by OpenSim's API). Most importantly, it treats physiological models of muscles as first class citizens alongside rigid bodies and kinematic joints, allowing to use OpenSim's native XML configuration file format to specify the structure and properties of muscle-tendon systems, which are created on top of Gazebo models specified in Gazebo's own file format (SDF), i.e., using Gazebo for physical modeling and OpenSim for muscle modeling, as exemplified in [34].

Bibliography

[1] Ivaldi, Serena et al. "Tools for simulating humanoid robot dynamics: A survey based on user feedback". In: 2015 (Nov. 2014). DOI: 10.1109/HUMANOIDS.2014.7041462.

[2] Open Source Robotics Foundation. *Robot Operating System (ROS)*. URL: http://www.ros.org/.

[3] Straszheim, Troy et al. *Conceptual overview of ROS catkin*. URL: http://wiki.ros.org/catkin/conceptual_overview.

[4] Faust, Josh and Pradeep, Vijay. *ROS message filters package (message_filters)*. URL: http://wiki.ros.org/message_filters.

[5] Faconti, Davide. *ROS PlotJuggler package (plotjuggler)*. URL: http://wiki.ros.org/plotjuggler.

[6] Meeussen, Wim. *ROS Control package (ros_control)*. URL: http://wiki.ros.org/ros_control.

[7] Open Source Robotics Foundation. *ROS Control in Gazebo*. URL: http://gazebosim.org/tutorials/?tut=ros_control.

[8] Franka Emika GmbH. *Franka Control Interface (FCI)*. URL: https://frankaemika.github.io.

[9] Franka Emika GmbH. *Franka Panda requirements*. URL: https://frankaemika.github.io/docs/requirements.

[10] Chitta, Sachin et al. "ros_control: A generic and simple control framework for ROS". In: *The Journal of Open Source Software* (2017). DOI: 10.21105/joss.00456. URL: http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf.

[11] Franka Emika GmbH. *Franka ROS Documentation*. URL: https://frankaemika.github.io/docs/franka_ros.

[12] Meeussen, Wim. *ROS Control: Writing a new controller*. URL: http://wiki.ros.org/ros_control/Tutorials/Writing%20a%20new%20controller.

[13] Franka Emika GmbH. *Franka Panda Joint Limits (Necessary Conditions)*. URL: https://frankaemika.github.io/docs/control_parameters.html.

[14] NaturalPoint, Inc. *OptiTrack NatNet SDK*. URL: https://optitrack.com/products/natnet-sdk/.

[15] Foote, Tully. *ROS geometry messages package (geometry_msgs)*. URL: wiki.ros.org/geometry_msgs.

[16] Martin Androvich. *Video: Franka Panda robot (simulated vs. real)*. URL: https://www.youtube.com/watch?v=cy7ZICqaFVQ.

[17] Open Source Robotics Foundation. *Gazebo simulator*. URL: http://www.gazebosim.org/.

[18] Franka Emika GmbH. *ROS Franka robot description package*. URL: http://wiki.ros.org/franka_description.

[19] Glaser, Stuart, Woodall, William, and Haschke, Robert. *ROS xacro package*. URL: http://wiki.ros.org/xacro.

[20] Open Source Robotics Foundation. *Using a URDF in Gazebo*. URL: gazebosim.org/tutorials?tut=ros_urdf.

[21] Gaz, Claudio et al. "Dynamic Identification of the Franka Emika Panda Robot With Retrieval of Feasible Parameters Using Penalty-Based Optimization". In: *IEEE Robotics and Automation Letters* 4.4 (Oct. 2019), pp. 4147–4154. DOI: 10.1109/LRA.2019.2931248. URL: https://hal.inria.fr/hal-02265293.

[22] Gaz, Claudio et al. *Estimated Franka Emika Panda Dynamic Parameters*. URL: https://github.com/marcocognetti/FrankaEmikaPandaDynModel/blob/master/pdf/RA-L_2019_PandaDynIdent_SUPPLEMENTARY_MATERIAL.pdf.

[23] Lynch, K.M. and Park, F.C. *Modern Robotics*. Cambridge University Press, 2017. ISBN: 9781107156302. URL: https://books.google.dk/books?id=5NzFDgAAQBAJ.

[24] Open Source Robotics Foundation. *Overview of ROS integration into Gazebo*. URL: http://gazebosim.org/tutorials?tut=ros_overview&cat=connect_ros.

[25] Tsouroukdissian, Adolfo Rodriguez. *ROS joint trajectory controller package*. URL: http://wiki.ros.org/joint_trajectory_controller.

[26] Siciliano, Bruno et al. *Robotics*. Springer London, 2009. DOI: 10.1007/978-1-84628-642-1. URL: https://doi.org/10.1007%2F978-1-84628-642-1.

[27] Open RObot Control Software (Orocos). *Orocos Kinematics and Dynamics Library*. URL: https://www.orocos.org/kdl.

[28] Glaser, Stuart. *ROS realtime tools package (realtime_tools)*. URL: http://wiki.ros.org/realtime_tools.

[29] Martin Androvich. *Video: Franka Panda robot (simulated vs. real)*. URL: https://www.youtube.com/watch?v=yX-IEuFacAU.

[30] Simbios. *OpenSim Model*. URL: https://simtk-confluence.stanford.edu:8443/display/OpenSim/OpenSim+Models.

[31] Martin Androvich. *Video: Franka Panda robot (simulated vs. real)*. URL: https://www.youtube.com/watch?v=cy7ZICqaFVQ.

[32] *OpenSim Forum: Regarding the possibility of forward dynamic simulations of robots in OpenSim 4.1*. URL: https://simtk.org/plugins/phpBB/viewtopicPhpbb.php?f=91&t=11947&p=34424.

[33] Neurorobotics Platform, Human Brain Project. *OpenSim support in the Neurorobotics platform*. URL: https://hbpneuroroboticsblog.wordpress.com/2017/09/20/opensim-support-in-the-neurorobotics-platform/.

[34] Neurorobotics Platform, Human Brain Project. *NRP-Opensim muscle modeling*. URL: https://developer.humanbrainproject.eu/docs/projects/HBP%20Neurorobotics%20Platform/2.0/nrp/tutorials/opensim_muscles/tutorial.html.