Compliant Humanoid Robots: Simulation and Control

Alessio Rocchi

11 November, 2015
Version: Draft

Università di Genova

CleanThesis

Advanced Robotics Department
Istituto Italiano di Tecnologia
Clean Thesis Group (CTG)

Robust Control Framework and Simulation Techniques for Compliant Humanoid Robots with application to the Darpa Robotics Challenge

Compliant Humanoid Robots: Simulation and Control

Alessio Rocchi

1. Reviewer Nikos G. Tsagarakis
Advanced Robotics Department
Istituto Italiano di Tecnologia

2. Reviewer Antonio Bicchi
Centro E. Piaggio
Università di Pisa

Supervisors Nikos G. Tsagarakis and Antonio Bicchi

11 November, 2015

Alessio Rocchi
Compliant Humanoid Robots:
Simulation and Control
Robust Control Framework and Simulation Techniques for Compliant Humanoid Robots with application to the Darpa Robotics Challenge, 11 November, 2015
Reviewers: Nikos G. Tsagarakis and Antonio Bicchi
Supervisors: Nikos G. Tsagarakis and Antonio Bicchi

Università di Genova
Clean Thesis Group (CTG)
Istituto Italiano di Tecnologia
Advanced Robotics Department
via Morego 30
16163 and Genova

Abstract

In the last years, a revolution took by storm the robotics community. Large round of investments have pushed the boundaries of applied robotics in different fields: autonomous car driving in unstructured environments as fostered by the DARPA challenge first, humanoids working in disaster scenarios through the DARPA robotics challenge (DRC), a new round of big investments by the US government and Toyota in autonomous car driving. This work documents several aspects conducted in research and engineering in the frame of the WALK-MAN team and as part of the DARPA robotics challenge effort. A whole-body control library will be introduced, together with details of the whole-body software infrastucture, starting from the simulator up to the middle and high level control. Details of the robot simulator will be covered, furthermore a novel dynamic simulator for compliant underactuated hands is proposed, with results on fidelity and robustness. A report on the experimental results of the WALK-MAN robot as used during the DRC finals will also be presented.

Contents

1 Introduction

In the last years, a revolution took by the storm the robotics community. Large rounds of investments have pushed the boundaries of applied robotics in different fields: autonomous car driving in unstructured environments as fostered by the DARPA challenge first, humanoids working in disaster scenarios through the DARPA robotics challenge (DRC), a new round of big investments by the US government and Toyota in autonomous car driving.

In particular, in the field of humanoid robotics, the DRC finals saw the partecipation of 25 teams [**DRC-what-happened**], with entries presenting their own robot built from the ground up for the competition, while others only focusing on the software development for algorithms in the fields of low and high level control [**beeson15**; **feng2015-rj**; **feng2015-oj**], planning, sensing and artificial intelligence, as well as teleoperation interfaces, transmission on degraded channels for teleoperation and sensing [**fallon2015-ni**], automated testing infrastractures, control frameworks, and simulation techniques for fast and robust simulation.

As part of the FP7 european project WALK-MAN, the WALK-MAN team has been funded to speed up the progress on the project in order to partecipate in the important DRC competition as an early demonstrator of the capabilities of the WALK-MAN robot and its infrastracture.

Part of the team took care of designing the robot, with the simulator in the loop in order to check the design against the task requirements set forth by the WALK-MAN project and by the DRC, a new SEA joint design has been created and a new network architecture for low-level control has been developed, based on ETHERCAT.

At the same time, the efforts of the iCub facility in the development of the YARP intrastructure and the iCub project have been integrated with the COMAN robot first, and the WALK-MAN robot afterwards. The *robotInterface* have been integrated and tested together with the partners at iCub facility to interface the low level control algorithms in the control boards of the two robots with the YARP interfaces for high level control.

Meanwhile, *YARP plugins for the Gazebo simulator* have been developed, in order to provide an easy to use framework where task developers could test their code prior to use the robot, a precious resource which needed to be carefully assigned to all the subgroups in the WALK-MAN team, as well as the other researchers in the department that needed to perform research work on the robot. At the same time, the models for the robot have been developed so that to be compatible with the new simulator and with the growing codebase of the *iDynTree* kinematics and dynamics library, providing kinematics and dynamics quantities for the robot to be used by the tasks developers. Both the models, simulator and the high level YARP interfaces

have been designed and configured in order to allow the testing and development of algorithms incrementally, starting from just the upper or lower body of the robot in order to allow manipulation and locomotion groups to work independently in the first part of development to maximize productivity.
In the meantime, the whole body framework *OpenSoT* has been developed in order to provide an high level task specification framework that could exploit the full capabilities of the robot in a whole-body fashion. From the developmenf of OpenSoT stemmed a set of tools that include task previewing, simplified geometries computation for collision detection, logging and plotting frameworks at the task and data level, and interfaces for different programming language for easy task prototyping. Together with the framework, high level control and modeling libraries, *iDynUtils* and *RobotUtils* were developed to integrate and easily provide functionalities based off popular robotics libraries (e.g. PCL, iDynTree, moveit) and build new ones such as dead reckoning based on the forward kinematics of the robot, whole body interaction with the environment by using anchor points to execute motions in intermittent multi-contact scenarios (such as the rising up motion that was part of the qualification videos for the team).
All these fundamental libraries have been subject to intensive testing, in the agile development mindset that characterized the development work of a big team, so that extensive unit testing and integration testing contributed fundamentally to establish sane regression tests practice during development and especially in the integration phases. Together with these tools, a build system has been setup together with iCub facility to ease the development of an always growing codebase with complicated interdependencies, resulting in the establishment of a so-called *superbuild* system that allowed new developers to easily join in, by downloading, compiling or installing all the parts of the WALK-MAN software ecosystem in less than one hour.
A component model, *GYM* has been developed in the meantime to streamline module development, with features like easy configuration managament, standardized commands for interaction with modules, graphical tools for plotting and on-line gain and parameters tuning for the algorithms and the modules containing them, a standardized state machine, communication channels for working on degraded networks.
At the same time, a teleoperation graphical interface, the pilot interface *PI* has been developed in order to provide graphical interfaces to all the modules to enable fast, practical and reactive teleoperation by the robot pilot team.
While the infrastracture was being built, development on the basic modules progressed. While it took more than 4 months to develop first version of the valve turning and locomotion modules, it took less than 2 months to develop the door opening task, and around one month for the car driving task, the drilling/one hand valve opening task. This can be attributed to the growing maturity of the infrastructure which allowed faster developmenf times while taking advantage of mature high level control schemes. Furthermore, the architecture allowed to switch from the

COMAN robot to the WALK-MAN robot with minimum effort, allowing to focus on the practical issues of debugging the new hardware, on the mechanical/electric and computational/networking aspects.
In the following thesis, elements of this work will be described in more depth, and in particular some details will be given about the simulation software, the high level whole-body control framework OpenSoT, and the software infrastracture developed.

1.1 Simulation

Simulation techniques for modern robot hardware could provide invaluable tools for design, research, and development for robot controllers. The DARPA Robotics Challenge (DRC), for example, fostered significant investment in reliable simulation tools for humanoid robots [HP14], which allowed teams to compete virtually (the Virtual Robotics Challenge, VRC) before qualifying for expensive robot hardware. Gazebo [**Koenig08**] is one of the most popular general-purpose robot simulators, funded through DRC efforts, but there are several others including V-REP and Webots. A few robot simulators and toolkits are specialized in grasping, such as GraspIt! [MA04], OpenRave [DK08] and OpenGRASP [Leó+10], which have built-in functionality for grasp analysis. However, these prior methods assume fully and precisely actuated grippers.
In 2 we will present the work on the Gazebo simulator, which has been chosen as the WALK-MAN and COMAN simulation environment also in light of its role as the official simulator for the Virtual Robotics Challenge (VRC) competition. Results on robust simulation of underactuated compliant hands will be showed.

1.2 Control

The field of whole-body planning and control tries to find the solution to the problem of executing one or more task at the same time, while exploiting the capabilities of the entire body of redundant, floating-based robots in multi-contact scenarios with the environment. Resolved velocity control is a popular choice for high level robot controllers, with latest trends both in research and application making use of Quadratic Programming (QP) to obtain a locally optimal solution to the hierarchical constrained IK problem. The interest is in part justified by the need, for practical reasons, to have robust solutions which are hard constrained by the physical limits of the robots, such as joint, torque and self-collision avoidance limits. In 3 the whole-body control framework OpenSoT will be presented, with focus on the implemented resolved velocity control schemes, compliant kinematic control schemes, and force control through admittance control. In 4 more details will be provided on how the algorithms developed for simulation and whole-body control fit in a large-scale humanoid architecture developed for robust control of the humanoid robots at the *Department of Advanced Robotics* of the *Istituto Italiano di Tecnologia*.

Simulation of Compliant Robots

2

2.1 Simulating Compliant Humanoid Robots

Simulation techniques for modern robot hardware could provide invaluable tools for design, research, and development for robot controllers. For this reason, numerous efforts have been spent in developing robot simulators in the past two decades ([Iva+14]), and more in general, in developing the large ecosystem that make simulating possible and convenient, from ever more fast and accurate dynamic solver libraries, to graphical editors for robotic systems and environments, planner libraries and visualization tools.

2.1.1 Simulators as Abstractions to the Physical Hardware: YARP Plugins for the Gazebo Simulator

In the past years, robotics researchers have been developing several robotics frameworks, or middleware libraries, such as OpenRDK ([Cal+08]), YARP ([Met+06]) or ROS ([Qui+09]) in order to ease the creation of generic applications for robots and encourage code reuse. Some of these frameworks even enforce a component system or formal validation , finally bringing in the research field some of the advancements brought in the industrial field in mission-critical applications. The performance overhead introduced by these frameworks is balanced by the architectural benefits, for example they allow to build modular systems to be integrated in order to execute complex and unforeseen tasks.

On the same level, modern simulators put an emphasis on robot simulation as a complex system, which is based but goes beyond the simulation of multibody dynamical systems, for example by simulating sensors.

The current section will describe some notable technical details of the work [MH+14] where a set of plugins for the Gazebo simulator enables the interoperability between a robot controlled using the YARP framework, and Gazebo itself. These plugins allow applications written for a set of YARP-enabled robots to be run on the simulator with no changes. At the moment of writing, a non-comprehensive list of YARP-enabled robots includes COMAN, iCub and WALK-MAN developed at the Istituto Italiano di Tecnologia, and the HyDRA robot developed at the University of Tokyo. Our plugins have two main components: a YARP interface with the same API as the real robot interface, and a Gazebo plugin which handles simulated joints, encoders, IMUs, force/torque sensors and synchronization. The robot model is provided to the simulator by means of an SDF file, an XML file that models the robots and its environemtn, by describing all the geometric, dynamic and visual characteristics of a robot, favouring a descriptive rather than a minimum set of parameters. Converters from Denavit-Hartenberg (DH, minal parameters representation) parameters to

Universal Robot Description Format (URDF, maximal parameters representation) formats are available (). Once the SDF is read from Gazebo, the plugins are loaded and associated with the simulated robot model and the simulated world. Different modules for the robots developed at the IIT have been developed using Gazebo and the plugins as a testbed: joint impedance control plus gravity compensation, dual arm Cartesian control for manipulation tasks, dynamic walking, etc. In particular, the plugins have provided an invaluable testbench during the development of the high-level control modules to be using during the Darpa Robotics Challenge by the WALK-MAN robot. The work is available as open-source at the address `https://github.com/robotology/gazebo_yarp_plugins`.

A preliminary stage of simulation during development of both research and production code, has many benefits that increase as the cost of production and maintanance of the hardware increases. Moreso, the ability of seamlessly switching between simulator and robot is an important factor that helps reduce development time and is not prone to errors and artifacts introduced during porting, enabling validation by simulation. Also the simulator provides fundamental advantages, like the ability to simulate faster than real-time.

2.1.2 Related Work

The *Open Dynamics Engine* (ODE, [Smi00]) is one of the most widely used rigid body dynamics engine in robotics simulation. ODE simulates chains of rigid bodies connected and constrained by different types of joints. It has a built-in collision detection system and implements hard contacts using non-penetration constraint whenever two bodies collide. Beside the large number of project that use it, at the moment the development has been paused. *Bullet* ([Erw03]) is another dynamic engine. It implements different direct/inverse rigid body dynamic algorithms (eg. Featherstone articulated body algorithm, [Fea07]) as well as different solvers (eg. Mixed Linear Complementarity Problem, MLCP) and contact models. Bullet is used for a wide range of projects and its community is active and continues to improve it constantly.

OpenRAVE ([Dia10]), *Webots* ([Mic04]), *V-REP* ([Roh+13]) and OpenHRP ([Kan+04]) all use the aforementioned dynamics engine, or variations thereof, and differ itself in the offer of libraries that aid in modeling, planning, controlling simulated systems.

The most recent MuJoCo ([Tod+12]) offers a novel approach to simulating dynamic systems with contact, and performed well comparatively to other simulators in a set of benchmarks[**erez15**].

In particular, in the last years, we have been witness to the significant investment in resources to provide reliable simulation tools for humanoid robots [HP14] which has been fostered by the Darpa Robotics Challenge (DRC). The results of this investment has allowed teams to compete virtually before qualifying for expensive robot hardware. Gazebo [KH04] is one of the most popular general-purpose robot simulators, funded through DRC efforts. It is a generic multi-robot simulator, historically

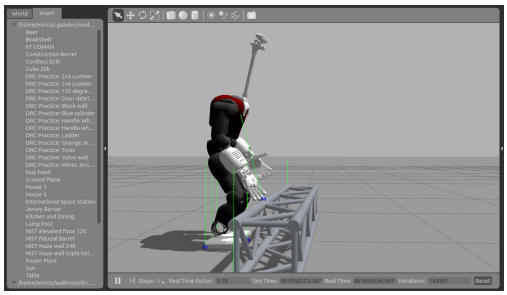
developed taking into account simulations in outdoor environments, and especially interesting nowadays for its high level of integration, as it provides realistic simulation of sensory feedback. including noise modeling. Other than ODE and Bullet, it allows to use DART [Tec13] and the promising SimBody [She+11], which is the only supported engine at the moment implementing variable step integration. For these reasons, and for its complex high level architecture, Gazebo has been used to compare algorithms for navigation and grasping in a controlled environment. To this day, its development is mainly performed by the Open Source Robotics Foundation (OSRF) under an open-source license, and is supported by a very large community. Gazebo is expandable by means of its plugin structure: thusly our YARP interface to the Gazebo simulator is a collection of plugins, the *gazebo_yarp_plugins*, that offer features wich will be described to various degrees of detail in the current section.

2.1.3 Gazebo plugins and YARP device drivers

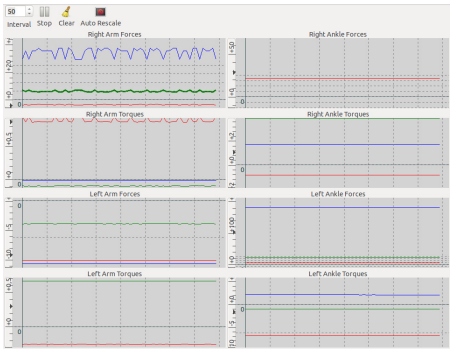
Gazebo plugins allow to extend the functionalities of Gazebo, while YARP device drivers are are used in YARP for abstracting the functionality of robot devices. *gazebo_yarp_plugins* are a set of plugins for Gazebo that allow to instantiate a set of YARP device drivers, in order to allow controlling a robot simulated in Gazebo via YARP. This abstraction is the fundamental step in allowing to control seamlessly a physical robot and a simulated one using the same interface, code, and ultimately, algorithms. The plugins/devices provided in the gazebo_yarp_plugins are the *Control Board*, *6-axis Force Torque sensor*, *Inertial Measurement Unit* (IMU) and a *Clock* plugin used for synchronization. The plugins stream data which is generated by the Gazebo simulator, and thus support all the features of the simulator, such as modelling Gaussian noise on the IMU readings. While the first three plugins are directly related to the simulated objects and sensors, the last one is a system plugin that synchronizes the YARP modules with the simulation time.

Control Board

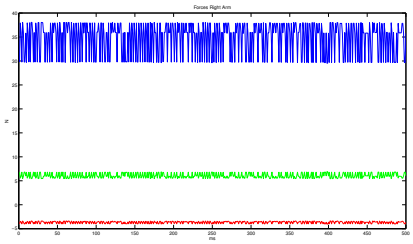
The Control Board plugin allows to control the robot using YARP Interfaces, it is implemented as a Gazebo Model plugin. Every control board allows the user to control one or more joints (a kinematic chain such as the arm or leg, etc.) as specified in a configuration file. For each controlled joint the control board opens different interfaces, permitting the use of different type of controllers for each joint. Such interfaces include position control, torque control, encoders reading, torque measurement and joint impedance control. Usually the number of instantiated control boards is equal to the number of kinematic chains. This duality between control boards and kinematic chains can be abandoned for the sake of performance on the real hardware, since every control board is usually implemented by a thread that streams data in a specified port. In that case, typical whole-body control then has to collate the network data coming from each control board and infer the global state of the robot. Each control board, during every cycle of simulation, reads



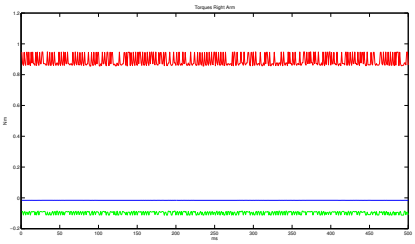
(a) Gazebo interface with COMAN



(b) A yarpscope showing data from Force/Torque sensors



(c) Plot of forces measured at the simulated Force/Torque sensor placed on the right arm. Forces along x,y and z are respectively in red, green and blue



(d) Plot of torques measured at the simulated Force/Torque sensor placed on the right arm. Torques along x,y and z are respectively in red, green and blue

Fig. 2.1: A Gazebo simulation running COMAN interacting with debris

position, velocity and torque values from the simulated joints and sends desired joints position or torques to the simulator. The values read from the simulator are broadcasted through YARP interfaces in the YARP network, in a similar way the desired joint values come from YARP interfaces (Figure 2.2). The following YARP interfaces are used to control the robot.

- **IPositionControl**: a position control with a linear trajectory generator considering a max joint speed
- **IPositionDirect**: a position control using Gazebo position PIDs
- **ITorqueControl**: a perfect torque follower
- **IImpedanceControl**: a joint impedance control with the following law

$$\tau_d = -P_d(q - q_d) - D_d\dot{q} + \tau_{offset} \tag{2.1}$$

where q_d is the desired equilibrium position, P_d is the desired joint stiffness and D_d is the desired joint damping. τ_{offset} is an extra term that can be used for gravity compensation or inverse dynamics control.

Furthermore, the Control Board implements the **IControlMode** interface that allows to change the type of controller online. All these interfaces are also available on the robot and they have the same behaviour.

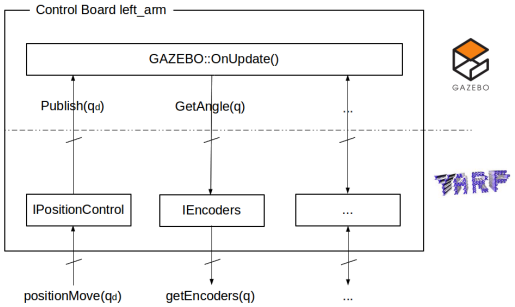


Fig. 2.2: Control Board plugin for the left_arm kinematic chain. yarp::IPositionControl interface has a method positionMove() that can be used to set joint values inside a YARP module. The plugin implements such interface by calling the Publish() method inside the Gazebo API to move the simulated joints at each OnUpdate().

6-axis Force/Torque sensor

A Force/Torque sensor measures a wrench in the robot structure (Figure 2.1). The sensor streams the data that are provided by Gazebo from a joint information. In fact, simulators like ODE use a maximal coordinate system, meaning that every body is simulated as an unconstraied ribid body (6DOFs), where constraints are enforced through the LCP formulation. This makes so that also joints are constraints, and since during simulation tipically the solver allows for constraint violation to a certain degree, which is proportional to the forces acting on the system and the equivalent stiffness of the constraint, such violation can be used to infer the forces and torques acting on the joint. On the YARP side, the reading of a generic sensor is implemented as a **IAnalogSensor** interface (Figure 2.3). The broadcasted data is a vector of six numbers representing the forces and the torques applied on that reference frame.

2.1.4 Real-Time Simulation of Robotic Systems: Clock

A fundamental aspect of simulation is the synchronization between the simulation code and the high level task code. Often, ad-hoc solutions emply a simulator-in-the-loop approach were after a certain number of simulation steps, and possibly each step, a control step is executed. In this case, we have true bidirectional synchronization between the simulation and the task code, that is, locked-step simulation. Since our code needs to be indipendent and agnostic of the simulator, it needs to be a networked application, and a synchronization mechanism needs to be put in place. In particular, the low level control algorithms specified in 2.2 is automatically synchronized with the simulation, and mimics the real-time characteristics of the low-level decentralized control running on the joint's boards of a robot. In order to simulate also real-time execution of the high-level code, synchronization between YARP modules and the simulated robot is needed. A YARP module is a process

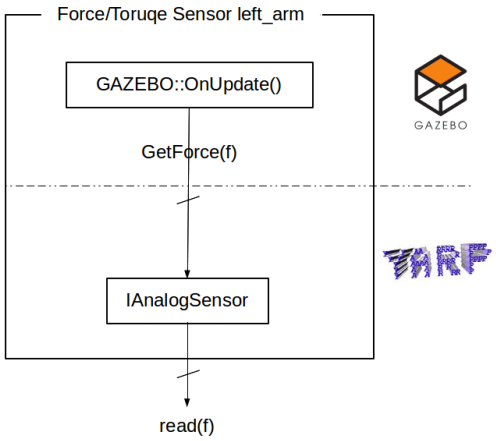


Fig. 2.3: The Force/Torque sensor in the left arm is implemented as a YARP IAnalogSensor interface. At every step the internal state of the plugin is updated with the last readings of forces and torques from the simulation.

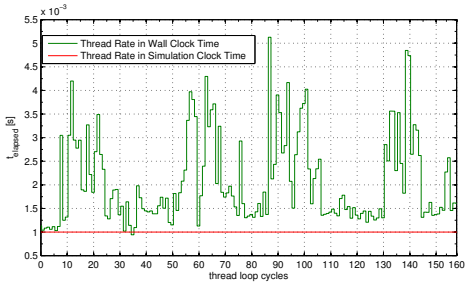


Fig. 2.4: Time elapsed between each execution of the control loop, measured in simulation clock time and in wall clock time. Desired thread rate is 1kHz and simulation time step is 1ms

in which one or more threads are started. When such modules are used in the real robot, the thread rate is timed by the machine (system) clock, also called the *wall clock*. In the same way, the client code's loop needs to be synchronized to the simulated clock when controlling a simulated robot. The *real-time factor* (*RTF*) of the simulation is given by

$$RTF = update\_frequency \times step\_time \tag{2.2}$$

and is kept to one when the desired update frequency is the inverse of the time increased at each step in the simulation. For instance if the simulation runs with a real time factor of 0.1, 10 seconds are needed to simulate 1 second of the physical system evolution. Within this situation, the controller process should also be slowed down 10 times to be coherent with the simulation. To solve this issue a *clock* plugin has been designed that synchronizes the high level control modules with the simulated time when controlling a simulated robot. The idea is similar to the ROS clock plugin for the Gazebo simulator, in that it is not a lock-step synchronization, meaning that in order to run the simulation faster than real-time we need the

control code to be fast enough, since the control code will be synchronized with the simulation clock but the simulation will not wait for the control code to complete execution before starting the next simulation step.

The *clock* plugin is implemented as a System plugin and publishes on a YARP port the time information from the simulator. For every simulation step, the simulation time is incremented and the timestamp is sent via socket. The data sent via this port acts as signal for a 1Khz simulated scheduler that wakes up the control threads that need to be woken up.

In general, all the YARP functions that provide access to the computer internal clock and support thread scheduling can be synchronized with an external clockof choice, be either coming from the simulator or other means (this is enabled with the *YARP_CLOCK* environment variable). Classes supporting periodic threads (*RFModule* and *RateThread*) are therefore automatically synchronized with the clock provided by the simulator. The *yarp::os::Time* functionalities are also transparently working using the wall-clock or the simulation clock depending on the environment variable. Thread sleeps are performed using the wall or simulated time depending on the circumstance.

More in detail, when synchronized with the simulation clock the *yarp::os::Time* delay does not explicitly sleep on a wall clock, rather a scheduler is synchronized with the simulation clock by performing blocking reads on the *YARP_CLOCK* port. This scheduler wakes up the threads that required a delay just once, when they have slept for the desired duration. Compared to the ROS::Time implementation which uses small sleeps on wall clock to check synchronization with the simulated clock, this allows to run simulations both slower and faster than real time and still have synchronization between threads and controls. In any case, when accessing the simulated clock Experiments showed the approach to be successful in synchronizing 1kHz control loops against simulations running 1kHz, thus having a 1ms clock granularity.

A similar solution for synchronization has been consequently used also in [Elj+14].

2.1.5 Notes on Simulation of compliant, highly redundant robots

The problem of simulating compliant humanoid robots in particular, and highly redundant robots in general, is the computation burden due to the large amount of joints (and thus, rigit bodies) that need to be simulated. Work is ongoing to properly simulate these systems on Gazebo, by using the Spong model and effectively doubling the number of simulated joints.While this approach allows to accurately simulate robot compliance but poses performance problems. A practical approach that has been used for the simulation of the robots developed at the Istituto Italiano di Tecnologia that are equipped with SEA joints, in particular when used in position control schemes, has been to tune the system PIDs to the actual joint compliance in the robot. Assuming a perfect controller on the physical robot, and taking into account that the simulated PID has a torque feedback, then the P gain

is dimensionally a stiffness, and the D gain is a damping. Technically, simulating compliant robots is easier than simulating stiff systems, were implicit integration schemes are needed for stability. The current Gazebo simulator uses a semi-implicit integration scheme, nonetheless simualaing very rigid robots (through very rigid PIDs) requires small integration steps, which compromise simulation speed. In our experiments, simulation is always run with a 1ms timestep.

2.1.6 Conclusions

In this section we have presented a set of Gazebo plugins, named *gazebo_yarp_plugins*, that allow to connect the robotics framework YARP to multi-purpose simulator Gazebo. Gazebo was chosen since it is easy to use, it has the possibility to switch between different rigid multi-body dynamics engines, it is Open-Source and has an active community. Our plugins are based on YARP device drivers in order to have the same interfaces in the real and simulated robot. This allows to write modules that will work both in the simulator and in the real robot without the need to modifiy the code. This is a very important paradigm in robotics since it minimizes the chance of introducing errors due to porting of the code, and allows for automatic validation by means of simulation. Furthermore the simulator becomes a tool that helps the developer in testing and validating before using the real platform. Such plugins consist in: a Control Board plugin to control the robot, a Force Torque sensor plugin and an IMU plugin. A special plugin dedicated to synchronization between modules and simulator was also implemented. The plugins were tested to simulate several humanoid bipedal robots, including the COMAN, iCub, and WALK-MAN from the Istituto Italiano di Tecnologia.

2.2 Simulating Grasps

Despite increasing popularity of compliant and underactuated hands, there exist few analytical and simulation tools for modeling such hardware. High-fidelity predictive tools are important in mechanism design as well as grasp planning and optimization to exploit the favorable features of compliant hands.

The following section will report studies [Roc+16] on a variety of simulation techniques to predict the success of a compliant gripper on irregular objects. In particular, performances of a new simulator will be presented, that integrates compliance simulation with a recent Boundary Layer Expanded Mesh (BLEM) technique for enhancing stability of contact normal and penetration depth estimation. The novel simulator is compared against existing ones using a set of stability and fidelity indices: *contact force smoothness* and *contact position and normal stability*. Scores along these indices are correlated with the simulator's accuracy of predicting the success/failure of a given grasp pose and preshape. A testing set of 13 grasps of varying success rate on physical hands were manually generated for the *RightHand Robotics Reflex Hand* and *Pisa-IIT Soft Hand* on 4 objects with a known 3D model and mass distribution. Each grasp is simulated using multiple techniques, and experiments find that the novel

simulator leads to improvements both in the stability indices, predictability of grasp success, and reduction of simulation artifacts.

The critical point of the evaluation lies on defining appropriate metrics, which can be considered universal in assessing the capabilities of a grap simulator to predit the behavior of compliant grippers grasping rigid objects. The physical and simulated experiments also have been designed to both highlight the importance of contact stability and accuracy during grasping (e.g. picking grasp poses where object ridges and geometric qualities of the object influennced the grasp outcome), and the importance of correctly simulating compliance (e.g. scenarios where the objec lies on the table and the hand fingers need to slide on the surface to perform a cage grasp).

2.2.1 Stable Simulation of Underactuated and Compliant Hands

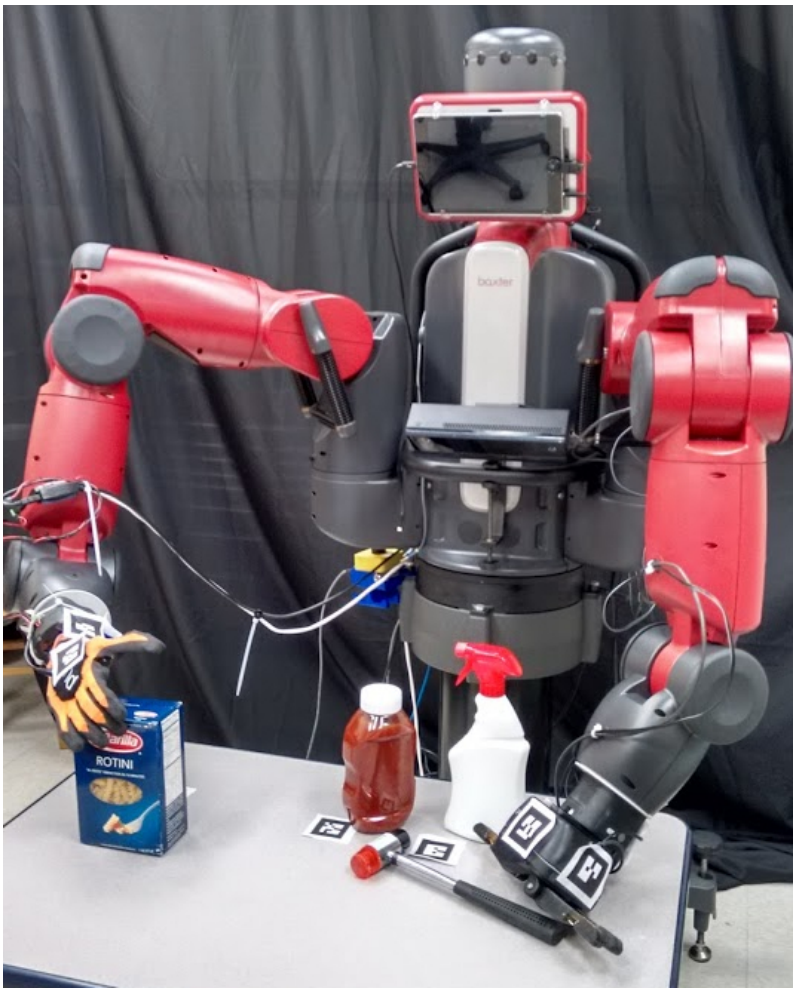


Fig. 2.5: the Baxter robot equipped with a Reflex Hand and a Soft Hand with the gasp set used for the experiments

An increasing number of robot hand designers have been moving towards devices that make use of compliance and underactuation. The iHY [Odh+14], Re-

flex Hand [Roba], Pisa/IIT SoftHand [Cat+14], Robotiq 3 fingers gripper [Robb], RBO [DB13] and RBO2 [DB14] and Yale Hands [Ma+13] are recent examples of underactuated compliant hands. The main benefits of such designs include hardware robustness and the capability of adapting the hand shape to a wide range of grasped objects. Underactuated hands provide many degrees of freedom of movement (DoFs) while being actuated by just a few degrees of actuation (DoAs), and clever passive mechanisms are designed to coordinate the actuation of multiple DoFs, which simplifies the otherwise complex hand control problem. Although compliant hands are built to be more robust to sensing uncertainties than rigid ones, they typically increase actuation uncertainties. As a result, successful grasping still requires deliberation, particularly for objects of irregular shape or mass distribution. It is also challenging to grasp objects in clutter because fingers may be disturbed by inadvertent contact with the environment or other objects. Hence, model-based predictions of motions and forces can offer powerful insights for hardware design as well as grasp optimization and planning.

Unfortunately, classical models for grasp analysis are poor at predicting the behavior of compliant hands. Classic kinetostatic analysis and kinematic simulation under the quasi-static assumption are inappropriate for studying the deformation of the hand under varying contact forces. Several researchers have turned to physics-based simulation analysis [Kap+15; Kim+13], which improves grasp predictability by evaluating hundreds or thousands of trials, in which the simulator explores the range of nondeterministic effects, e.g., external perturbations, pose uncertainty, geometric uncertainty, and joint friction. Reliable simulation is essential for such studies. However, compliant hands pose a challenge for dynamic simulation tools, which often have trouble remaining stable in the presence of multiple frictional (and sometimes sliding or rolling) contacts and stiff internal mechanisms, like springs and tendons. When the object is not considered to be fixed to the environment, object-hand interactions need to be simulated taking dynamics into account, and a faithful representation of contact has to be available in order to detect the making/breaking of contacts, inadvertent contact, and sticking/slipping conditions.

This paper presents a new physics simulator for compliant hands interacting with rigid objects. It integrates models of compliant elements with recent contributions in robust mesh-mesh contact generation methods. Compliant elements are modeled as spring-dampers while stiffer elements like tendons are handled using constraints and Baumgarte stabilization. To handle mesh-mesh contact, the recent *Boundary Layer Expansion Mesh BLEM* method is used, that avoids many simulation artifacts by generating continuously differentiable contact points [Hau13]. To handle large number of contacts, contact clustering schemes is employed. Various variations of BLEM are explored against existing contact detection schemes in the popular Open Dynamics Engine [Smi00] and Bullet [Erw03] physics engines. This comparison is particularly meaningful as those same contact detection schemes are used in vast the majority of the robot simulators software introduced in 2.1.2 with the exception

of MuJoCo and SimBody. Two contact filtering schemes — *k-means clustering* and *hierarchical clustering* — are then compared against the *contact sorting* method implemented in the Gazebo simulator.

Finally, a set of indices is established for characterizing the stability of grasping simulations at the level of contact point and force prediction. A series of experiments is performed, comparing grasps performed on the physical hardware against results from a variety of simulation tools. Experiments demonstrate that grasp predictability is found to be correlated to the contact stability indices. Furthermore, the new simulator is demonstrated to achieve improved fidelity regarding the predictability of grasp success/failure compared to existing off-the-shelf simulators.

2.2.2 Related Work

Robot Simulators. A few robot simulators and toolkits are specialized in grasping, such as GraspIt! [MA04], OpenRave [DK08] and OpenGRASP [Leó+10], which have built-in functionality for grasp analysis. However, replicability of these prior methods assume complete, precise actuation of the hand.

In [Bon+14] a dynamic simulation of the Pisa/IIT Softhand is implemented in the multi-body dynamics simulator MSC Adams [Cor]. A method to generate pre-grasp palm configurations w.r.t. the object pose is provided. The simulator demonstrated moderate fidelity to an experimental scenario, but the authors acknowledge difficulties with hand-object penetrations and estimation of contact normals.

A recent contribution in the Klamp't simulator is the notion of boundary-layer expanded meshes (BLEM) which were demonstrated to enable robust mesh-to-mesh contacts, even in the presence of non-watertight meshes [Hau13]. In this work we extend Klamp't to handle compliant mechanisms, and also we adapted the BLEM technique into Gazebo as a plugin.

Grasp analysis and planning. Classical grasp analysis typically studies the shape of the grasp wrench space (GWS), which is the convex hull of wrenches applicable by a unit force at each contact. For example, the *epsilon quality* ε_{GWS} [FC92; PK13] studies the size of the largest ball inscribed within the GWS, which is a proxy for how likely the grasp will resist a random disturbance force. But recent studies have suggested weaknesses of classical methods, such as an inability to measure robustness to perturbations in contact locations or grasp / object pose [WA12]. Physics simulation in grasp quality assessment has been shown to yield improved prediction over classical criteria [Kap+15; Kim+13]. In [Kim+13] the robustness of automatically generated grasp sets is assessed, and success of grasps in physical experiments is correlated with the predicted success using simulation. In [Kap+15] grasp stability criteria are correlated with grasp success as predicted by humans on a large database of objects. They concludes that physics-based metrics are more consistently predictive than a classical metric. The technique presented in this paper extends physics-based grasp assessment to the case of underactuated and compliant hands.

Simulation quality metrics. Generic criteria for evaluating the contact response fidelity of dynamic physics engines were established in [BB07], such as accuracy of the friction coefficient, and penetration error. [Ere+15] defines some application specific metrics, like a grasping metric, in which robustness to the contact handling is tested by increasing the integration step until the simulated grasp becomes unstable. In this work instead, a correlation between micro-indices of contact stability in the physics to the macroscopic fidelity of the simulator with respect to a physically attempted grasp will be highlighted.

2.2.3 Stable Dynamic Grasp Simulation

An underactuated hand is modeled as a mechanism composed of articulated rigid links with n degrees of freedom and n_a degrees of actuation, with $n_a < n$. The state of the fingers is denoted as $q \in \mathbb{R}^n$. A control $u \in \mathbb{R}^{n_a}$ gives rise to a net torque on the joints $\tau \equiv \tau(q, u) \in \mathbb{R}^n$ which summarizes the sum of internal torques including gearing, stiffness, damping, joint stops, and friction.

For the three-fingered ReFlex, $n_a = 4$, with 1 DoA for each finger. An additional pregrasp mechanism changes the direction in which finger 2 and 3 close from power grasp to pinch grasp. Assuming the distal joints rotate along a fixed axis, $n = 7$, but in general the soft joint between the proximal and distal joints may flex and stretch. For the five-fingered SoftHand, $n_a = 1$, $n = 19$.

When in contact with external objects, the pressure distribution on a robot's link is a function $\rho_i(x) : \partial S_i(q) \to \mathbb{R}^3$ where $\partial S_i(q)$ denotes the surface of link i. Given such a distribution, the resultant torques on the robot's joints are

$$\tau_{contact} = \sum_i \int_{x \in S_i(q)} J_x^T(q) \rho_i(x) dx \tag{2.3}$$

where $J_x(q)$ is the Jacobian matrix of point x.

Thus, the dynamics of the robot in contact are given by

$$B(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = \tau(q, u) + \tau_{contact} + \tau_{ext} \tag{2.4}$$

where $B(q)$ is the robot's mass matrix, $C(q, \dot{q})$ is the Coriolis force matrix, $G(q)$ is the generalized gravity vector, and τ_{ext} denotes torques resulting from external forces, such as joint friction.

A simulator is given an initial state $(q_0, \dot{q}_0)$, a control trajectory $u(t)$, and a final time T. The goal is then to generate a trajectory of the robot $q(t) : [0, T] \to \mathbb{R}^n$ as well as the motions of other objects $O_1, \ldots, O_m$. The most challenging problems in simulating underactuated hands are 1) calibrating accurate models, particularly of τ, 2) calculating contact force distributions ρ to prevent interpenetration and to simulate friction, and 3) maintaining stability under often stiff dynamics, particularly in τ.

The linear complementarity problem (LCP) [Ani97] method is a popular method for calculating contact force distributions in rigid body physics simulators because it solves for forces that prevent interpenetration (to some degree; slight interpenetration does occur due to numerical errors). All the simulators compared in this section use the LCP method, although other techniques such as penalty forces and impulse-based methods have also been developed.

Modeling underactuation and compliance

Underactuated and compliant hands are linked with transmission mechanisms, e.g., tendons or mechanical linkages, that distribute actuator effort across multiple joints. They also include spring mechanisms that restore the hand to a consistent rest state once gripping effort is removed. Simulations must allow for actuators to drive multiple links forward, but also to allow for forces on one link to affect the distribution of effort across other links. We use a fairly general model for these effects [Gri+12].

First, a general constraint model that relates actuator displacements s to configuration displacements q is used:

$$s = Rq \tag{2.5}$$

where the reference configuration is chosen so the zero actuator and joint correspond. The $n_A \times n$ matrix R determines how joint movements pull on each actuator, and can be constant, as in the case of the Pisa/IIT Softhand, or configuration dependent in the case of the ReFlex Hand [Bir11]. In practice the tendons may be slightly elastic, but it will be assumed that they are sufficiently stiff so that direct simulation is unstable.

It is also assumed the drive mechanism generates resultant torques on individual joints in order to maintain these constraints. Denote the force generated by the actuator as $\sigma \in \mathbb{R}^n$, and the torques generated by the drive mechanism be denoted $\tau_d \in \mathbb{R}^n$. By the principle of virtual work, we have $\tau_d = R^T\sigma$. Let us also define the joint torques produced by spring mechanisms as $\tau_s = -Eq$ where E is a $n \times n$ joint stiffness matrix (which is usually diagonal). Neglecting friction effects, the resultant vector of joint torques is

$$\tau = R^T\sigma - Eq. \tag{2.6}$$

Rather than simulate these torques directly, a rest state of q is derived that attains quasistatic equilibrium by equating τ with the sum of inertial effects minus contact forces in (2.4). First, the desired value τ is computed as determined by the l.h.s. of (2.4) minus external forces:

$$\tau \equiv B(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) - \tau_{contact} - \tau_{ext} \tag{2.7}$$

We can then solve for σ and q that satisfy the constraints (2.5) and (2.6) by solving a system of linear equations, which has a closed form solution in terms of s, R, E, and τ [Gri+12].

Given the solved q and σ, the joints are simulated using critically- or over-damped PID controllers with setpoint q. A possible strategy is also to simulate the effect of the actuator force σ on the actuator state s.

Note how this procedure needs to iterate over multiple time steps to achieve equilibrium between mechanism torques and contact forces, which may cause chattering especially if contact forces are nonsmooth. Future work may achieve better results by simultaneously choosing mechanism torques and forces via a constraint in the LCP solver.

2.2.4 Contact generation

Contact generation is an important stage in simulation when the region of contact between two bodies is approximated by a finite number of contact points and normals. In LCP solvers, contact forces at these points are restricted to be positive in the normal direction with tangential component limited by the friction coefficient. Moreover, residual penetrations caused by numerical error are corrected for via penetration depth estimates computed in this stage.

It is difficult to compute a stable, accurate representation of the penetration region between nonconvex bodies, and hence existing engines ODE and Bullet use the approximate GIMPACT and OPCODE methods for calculating mesh-mesh contact points. The experience of many users as well as recent work suggested this method often leads to unstable simulations [Hau13]. The same work presented the Boundary Layer Expanded Mesh (*BLEM*) representation that approximates a mesh with a slightly expanded version, which allows calculating accurate penetration distances and normals [Hau13].

The *BLEM* method treats a mesh as having a small margin r around it, and when two BLEMs are overlapping by a distance less than the sum of their margins, the penetration depth and direction can be computed by distance queries on the underlying meshes. More precisely, a BLEM(M, r) is the Minkowski sum (or dilation)

$$M \oplus S = \{v + s | v \in M, s \in S\} \tag{2.8}$$

with $S = B(0, r)$ being a sphere centered in the origin, with radius r.

Contact generation between two BLEMs (M_1, r_1) and (M_2, r_2) is handled by detecting all pairs of primitive triangles in M_1 and M_2 whose distance d is less than $r_1 + r_2$. A point is then generated in the overlapping region between the two closest points p_1 and p_2 on the respective triangles. The penetration distance is $r_1 + r_2 - d$ and the normal is a unit vector proportional to $p_2 - p_1$.

Contact clustering

During close mesh-mesh contact, the contact generator may produce a large number of contact points, which leads to slow computation of contact response. This is a particular problem for LCP-based solvers due to their superlinear running time. As a result, researchers have taken an interest in contact clustering methods that combine multiple contacts into fewer contacts, which still yield similar post-contact response. Also, since having nearby contacts lead to ill-conditioned wrench matrices, clustering may in fact improve numerical stability.

Our experiments indicate that a deterministic version of the k-means algorithm, applied to the 7-D space of positions, normals, and friction coefficients, provides more stable clusters than hierarchical clustering, or the simple contact sorting used in Gazebo (Fig. 2.6).

For example, a distribution of forces on a planar surface of a rigid body yields an equivalent wrench to a distribution on the convex hull of that surface. As a result, a set of coplanar contacts can be collapsed into their convex hull with zero loss of theoretical simulation accuracy in LCP-based solvers. Similarly, several contacts with similar position and normal can be approximated as a single "average" contact with only a small loss of accuracy.

In *k-means clustering*, the contact points are partitioned in a number of clusters equal to the maximum admissible number of contacts. The partitioning minimizes the sum of the distances between each contact point and the center of the clusters which will become the reduced set of contact points. In *hierarchical clustering*, clusters are determined so to obey a hierarchical subdivision and has the benefit over the *k-means clustering* of being a deterministic rather than an iterative algorithm [Rok10]. In *contact sorting*, the contacts are first sorted according to the penetration depth, and the contacts with the greatest associated penetration depth will be used to resolve contact forces. The first two methods are illustrated in Fig. 2.6.

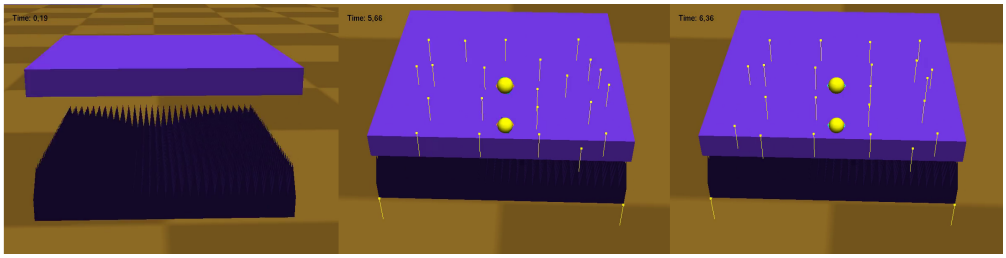


Fig. 2.6: Illustrating two contact clustering methods on a 2,000-spike "spiky block". Left: hierarchical clustering. Right: k-means clustering. Centers of pressure are draw

2.2.5 Contact stability metrics

In the following section are introduced the metrics used in this work for evaluating the stability of contact points and contact forces at a fine granularity during the simulation of grasping. These metrics correspond to the position, normal, and force

dimensions of point contact models used in classical grasp analysis, which forms the basis of most grasp planning techniques.

Contact Force Variation

At equilibrium, we expect the contact forces exerted on the objects to be as close to constant as possible. Our first measurement penalizes fluctuations and spikes in force at an (ostensible) equilibrium state.
The resultant contact force exerted on body i is $f_i \equiv \int_{x \in S_i(q)} \rho_i(x) dx$, which the simulator approximates by a finite set of contact forces. The contact force smoothness index measures the normalized standard deviation of resultant forces. (It may be possible to also include resultant torques $m_i \equiv \int_{x \in S_i(q)} (x - o_i) \times \rho_i(x) dx$ but these are highly correlated to forces for most grasps so we ignore them.)
Over a span of time t where the object should be held in equilibrium by the physical hand, we measure the quantities $\sqrt{\|Var[f_i(t)]\|_F} / E[\|f_i(t)\|]$ for all bodies i in the set C of bodies touching the object. Here $\|\cdot\|_F$ denotes the matrix Frobenius norm. We then report the final score (0 being best):

$$S_{cf} \equiv \frac{1}{|C|} \sum_{|i \in C|} \sqrt{\|Var[f_i(t)]\|_F} / E[\|f_i(t)\|]. \tag{2.9}$$

Contact position and normal variation

The second and third scores measure the stability of contact positions and normals during an equilibrium grasp. A simulator generates a set of point contacts at location $x_1, \ldots, x_k$ and their corresponding normals $n_1, \ldots, n_k$ (k varying by time step). The scores measure the fluctuations of the average contact position $\hat{x}_i$ on body i and average contact normal $\hat{n}_i$ over time (0 being best).
The equations are as follows:

$$S_{cp} \equiv \frac{1}{C} \sum_{|i \in C|} \sqrt{\|Var[\hat{x}_i(t)]\|_F} \tag{2.10}$$

$$S_{cn} \equiv \frac{1}{C} \sum_{|i \in C|} \sqrt{\|Var[\hat{n}_i(t)]\|_F} \tag{2.11}$$

An example of these results for the spiky plane example is given in Fig. 2.7.

2.2.6 Experiments

Experiments are performed both with a RightHand Robotics' ReFlex Hand and Pisa/IIT SoftHand. A set of grasps with varying degrees of robustness are designed by human inspection and tested on a set of physical objects and their simulated counterparts.
For the two hands, a series of grasps are selected for each object in the set: *hammer, ketchup bottle, spray bottle* and a *pasta box*. The *hammer* and the *pasta box* are

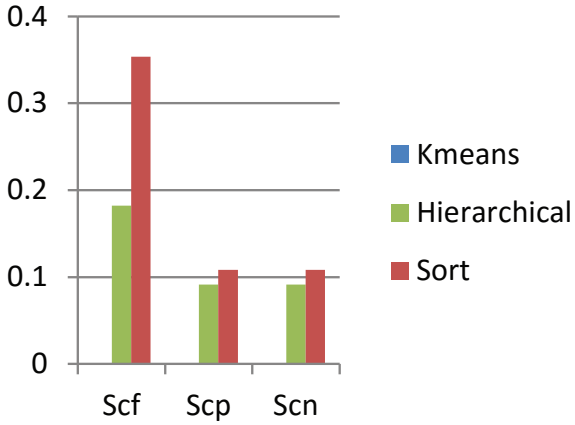


Fig. 2.7: Contact stability metrics S_{cf}, S_{cp}, and S_{cn} over the spiky plane example of Fig. 2.6 for various clustering methods. K-means leads to almost imperceptible instability ($1E-17$ for all three indices)

Object	Weight	Size
Hammer	480 g	290mmX35mmX95mm
Spray Bottle (no cap)	703.5 g	220mmX120mmX40mm
Spray Bottle (with cap)	726 g	270mmX120mmX40mm
Ketchup	948.6 g	195mmX100mmX70mm
Pasta Box	487g	120mmX19mmX72mm

Tab. 2.1: Weight of all the objects and approximate dimensions

modeled by hand. The rest of the objects have been scanned using a *Makerbot Digitizer*. The results and details of the mesh reconstruction are shown and detailed in Fig. 2.8. Dimensions are given in Tab. 2.1. We assume an approximate mass distribution for the *hammer*, and for the mass of the the rest of the objects. In all the experiments, the objects are laying on a table.

Human-chosen grasps are selected in order to have a selection of robust grasps, non-robust grasps, and failed grasps as described in Tab. 2.2. Robust grasps tolerate small variations in the object pose and disturbance forces. Non-robust grasps succeed some of the time, but do not always tolerate such variations. Failed grasps fail consistently. The hands and the table are marked with fiducial markers, and their position and orientation are recorded for each grasp via a Microsoft Kinect camera. The objects are placed on the table in a known pose, and the grasps illustrated in Fig. 2.12, 2.13 and 2.14 are performed. For every nominal grasp pose, the hand has been positioned manually on the desired configuration via the gravity compensation control of the *Baxter* robot. A set of 6 trials is performed for each grasp, and grasp scores are averaged over the attempts over three indices:

- $S1$ **object pose deviation**. This is designed in order to be performed easily by eye without complicated measurements attempts. As it can be seen in 2.10 it is practically useful only in the *hammer* set of grasps, because of the

Object	Pose	Effort	Grasp Position	Preshape
Hammer	P1	0.3	Handle	0
Hammer	P2	0.3	CoG	0
Hammer	P3	0.3	Head	0.5
Spray Bottle (no cap)	P1	0.2	Low (spray facing outside)	0
Spray Bottle (no cap)	P2	0.2	High (spray facing outside)	0
Spray Bottle (no cap)	P3	0.2	Low (spray facing inside)	0
Spray Bottle (no cap)	P4	0.2	High (spray facing inside)	0
Spray Bottle (with cap)	P1	0.2	High	0
Ketchup	P1	0.3	Cap	1.5
Ketchup	P2	0.3	Under cap ridge	1.3
Ketchup	P3	0.2	Lateral ridge	0
Ketchup	P4	0.2	Above lateral ridge	0
Pasta Box	P1	N/A	Top, parallel	N/A
Pasta Box	P2	N/A	Top, tilt along yaw	N/A
Pasta Box	P3	N/A	Side	N/A

Tab. 2.2: Grasp configuration

advantageous lever the CoM has on the grasp. In the bidimensional case, the index uniformly discretizes the possible swing angle of the hammer in 3 regions, with scores 1, 0.6 and 0.3. The score will be 0 in case of an unsuccessful grasp.

- $S2$ **the normalized number of fingers in contact**. For the Reflex Hand, the index can assume values 1 for a three-fingers grasp and 0.5 for a two fingers grasp. For the Soft Hand, the index can assume values 1,0.75,0.5 and 0.25.

- $S3$ **ability to resist perturbation**. By applying a shaking motion on the hand of the robot, we check if one of the previous indices changes in value as a consequence of the perturbation, in which case the score will be 0.6. In case of a dropped object as a consequence of the perturbation, the score will be 0.3.

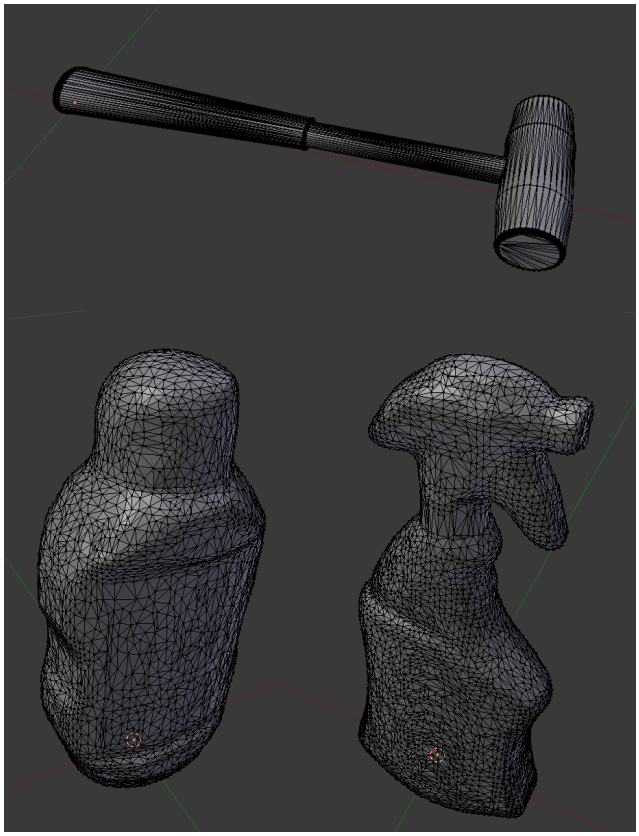


Fig. 2.8: The 3d laser range scans are reconstructed using Poisson surface reconstruction, and the number of faces of the mesh is then reduced using Quadric Edge Collapse Decimation. All the meshes have a number of faces between 8000 and 12000. As a last step, the bottom of the object is manually cropped to a flat surface.

Fig. 2.9: comparison side by side of the real objects and the 3d scanned

The grasp procedure for the *Reflex Hand* consists in simultaneously closing each finger until the motor's effort threshold is met, at which point the finger is stopped. For the *Soft Hand*, the fingers close simultaneously until the nominal torque limit of the single actuator is reached. To check if the grasp is successful, the object is then raised. If successfully raised, the robustness of the grasp is checked by shaking the gripper by hand.

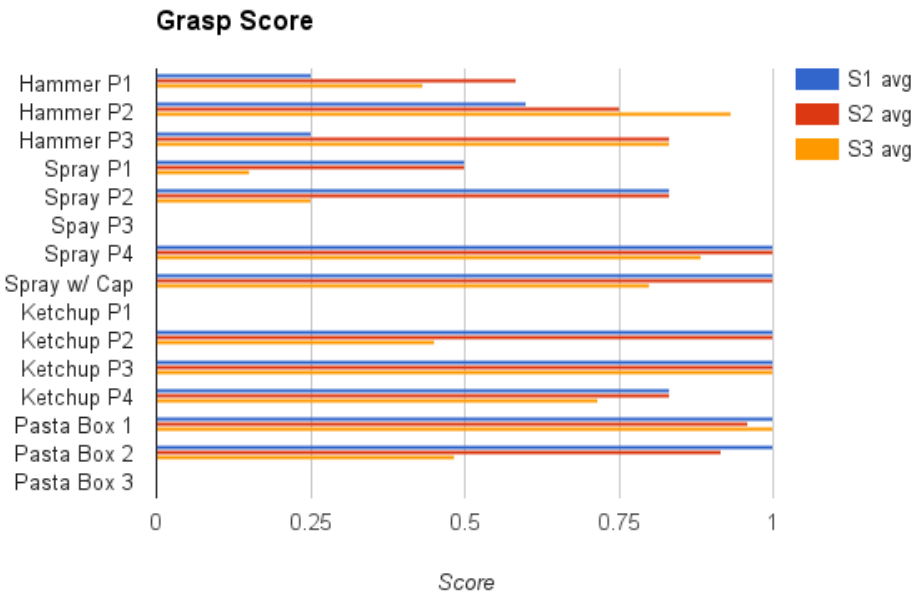


Fig. 2.10: Grasp scores for physical experiments. On the x axis, $1-3$ are for *hammer-P1* to *hammer-P3*, $4-7$ are for *spray-P1* to *spray-P4*, while score 8 is for *spray-full-P1*, $9-13$ are for *Ketchup-P1* to *Ketchup-P5*

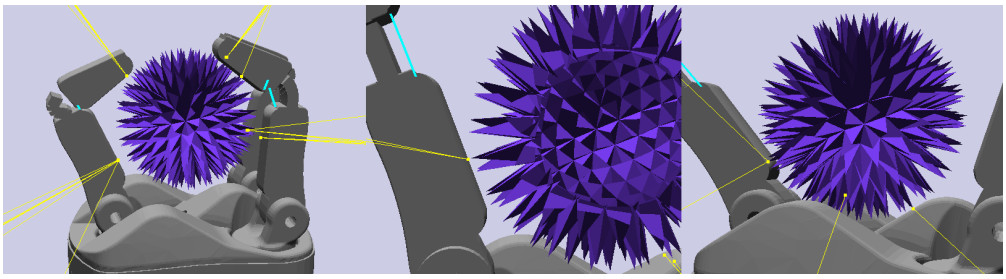


Fig. 2.11: Since the *BLEM* method works by expanding the meshes via Minkowski dilation, a preshrink method was introduced so that the boundary layer added to a preshrunk mesh will sum up to the approximate size of the original mesh. In figure, from left to right, a $5mm$ boundary layer shows gaps between the hand and the object, a $1mm$ boundary layer, and a $1mm$ layer with preshrink.

The grasp is then performed in simulation. We simulate the *Reflex Hand* using *Klamp't* v0.6.2, and the *Soft Hand* using *Gazebo* v4.0 with a preliminary version of the *Soft Hand plugin* [Ros]. The *Gazebo* simulator is tested against our patch [Roca] where *BLEM* is used. In Gazebo, the simple contact sorting algorithm is used for contact filtering. The *Klamp't* simulator is tested with *BLEM* against the default in Open Dynamics Engine *OPCODE*. In Klamp't examples, we use k-means clustering with a maximum of 20 contacts.

The robot's motor parameters, tendon Baumgarte stability constants, and friction parameters were tuned by hand so that the hands would move qualitatively similarly

to the real hands, and to be stable both in free space motion and while grasping on a spherical object. No quantitative calibration procedure was performed.

Fig. 2.12: Comparison between simulation and experiment: *Hammer-P1*, *Hammer-P2*, *Hammer-P3*

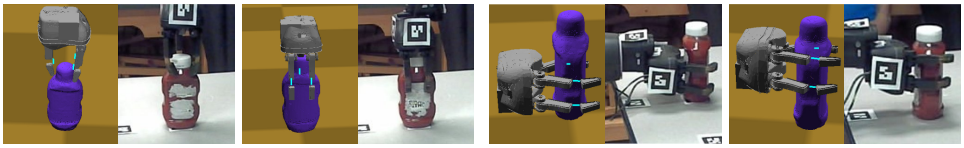


Fig. 2.13: Comparison between simulation and experiment: *Ketchup-P1*, *Ketchup-P2*, *Ketchup-P3*, *Ketchup-P4*

Fig. 2.14: Comparison between simulation and experiment: *Spray Bottle (with cap)*, *Spray Bottle (no cap) - P1*, *Spray Bottle (no cap) - P2*, *Spray Bottle (no cap) - P3*, *Spray Bottle (no cap) - P4*

Fig. 2.15: Comparison between simulation and experiment: *Pasta Box - P1*, *Pasta Box - P2*, and *Pasta Box - P3*.

The simulation procedure is as follows:

- The object and hands are spawned in the pregrasp and preshape configuration,
- As in the physical hardware, the hands are closed until an actuator force threshold is reached.
- The hand is lifted by 10cm,
- The grasp is perturbed by commanding jerking movements of the wrist,
- Finally the hand opens to drop the object.

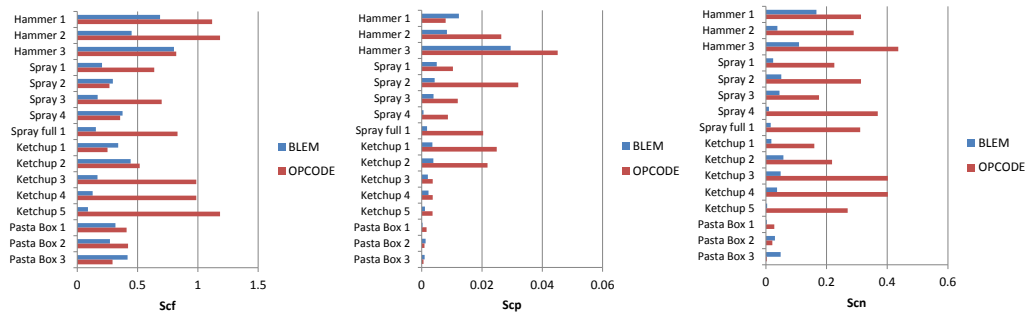


Fig. 2.16: Metrics S_{cf}, S_{cp}, and S_{cn} over the simulated grasp set. Lower numbers are better.

Fig. 2.16 illustrates the results of the contact stability indices between BLEM and OPCODE contact generation. It can be observed BLEM leads to higher stability in general, most strikingly for the contact normal metric.

Grasp	Grasped	Dropped	Grasped	Dropped
ReFlex	**ODE / BLEM**		**ODE / OPCODE**	
Hammer 1	Y	Y	Y	**N**
Hammer 2	Y	Y	Y	Y
Hammer 3	Y	Y	Y	**N**
Spray 1+	Y	Y	Y	**N**
Spray 2+	Y	Y	Y	**N**
Spray 3*	Y	Y	Y	Y
Spray 4	Y	Y	Y	**N**
Spray Full 1	Y	Y	Y	Y
Ketchup 1*	Y	Y	**N**	–
Ketchup 2	Y	Y	**N**	–
Ketchup 3	Y	Y	Y	**N**
Ketchup 4	Y	Y	Y	**N**
SoftHand	**Gazebo / BLEM**		**Gazebo / OPCODE**	
Pasta Box 1	Y	Y	X	Y
Pasta Box 2+	Y	Y	X	Y
Pasta Box 3*	N	–	N	–

Tab. 2.3: Success rates of ReFlex grasps (top) and SoftHand grasps (bottom) in simulation. Asterisks indicate failed grasps on physical experiments. The plus sign indicates non robust grasps on physical experiments. An entry of X indicates variability for the same initial conditions.

Tab. 2.3 illustrates the results of simulations. 12/12 successful grasps were reliably grasped by BLEM and 9/12 were grasped by OPCODE. In all successful grasps, the object was securely held through perturbations. In the two ODE/OPCODE failure cases, the hand penetrated completely through the object without contact. In the two Gazebo/OPCODE unreliable grasp cases, we observed nondeterminism in the simulator which caused the grasp to fail in approximately 1/3 of simulation trials, even with the exact same initial conditions.

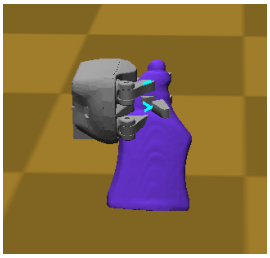


Fig. 2.17: An artifact exhibited in OPCODE causing the object to penetrate and then "stick" to the robot's finger.

Perhaps more worrisome is the result that ODE/OPCODE did not successfully release the grasped object in 7/9 successful grasps. This is explained by major interpenetration artifacts, in which the object gets "stuck" on the robot's finger and cannot be let go (Fig. 2.17.
BLEM did not successfully predict 2/3 failed grasps. This likely indicates an improperly calibrated coefficient of friction or absolute grasp force. This suggests that although stable simulation is crucial for obtaining reasonable predictions, it is still challenging for simulation to accurately discern the boundary of feasible/infeasible grasps without considerable tuning. An interesting area for future work would be to automatically tune the robot model to obtain a match with physical experiments.
Lastly, computation times during grasp for the *Klamp't* case were similar between BLEM, with 2.7ms and OPCODE, with ~ 2.9ms for each millisecond of simulated time. For Gazebo, the experimental BLEM patch resulted in computation times of ~ 2.4ms against ~ 1.6ms of OPCODE for each millisecond of simulated time. All measurements were taken on an Intel Core i5-4460 CPU @3.2HGz.

2.2.7 Conclusions

A series of simulation stability and fidelity criterion have been established. They have been tested on two different compliant and underactuated hardware platforms, the *RightHand Robotics' Reflex Hand* and the *Pisa/IIT SoftHand*. A set of grasps has been performed on the physical hardware using 4 different objects of different shapes and characteristics, in different grasping scenarios, and then validated using *Klamp't* and *Gazebo*, using different techniques for *contact clustering* and *contact generation*. The experimental results show that simulations performed with the *BLEM contact generation* algorithm and the *k-means clustering* contact filtering algorithm have the highest stability indices, and also exhibit the fewest artifacts.

Whole-Body Control

3

The field of whole-body planning and control tries to find the solution to the problem of executing one or more task at the same time, while exploiting the capabilities of the entire body of redundant, floating-based robots in multi-contact scenarios with the environment.

Several and different approaches are being investigated by the research community, such as optimization, inverse dynamics, inverse kinematics with dynamic filters and admittance control. .

Inverse Kinematics (IK) is a fundamental component for a vast number of application in robotics. In particular, kinematic control is still a popular choice for high level robot controllers. While the kinematic control problem has been tackled in literature using several different approaches, latest trends both in research and application make use of Quadratic Programming (QP) to obtain a locally optimal solution to the hierarchical constrained IK problem. The interest is in part justified by the need, for practical reasons, to have robust solutions which are hard constrained by the physical limits of the robots, such as joint, torque and self-collision avoidance limits. In this section the theoretical foundations of kinematic control based on QP optimization will be presented, and the formulation of a comprehensive set of tasks and constraints will be provided, according to the capabilities of the developed framework. The kinematic control, and thus the formulation for the tasks and the contraints in the framework will be expressed at the velocity level, meaning the solution of each step of the IK solver will return a displacement δq that can be actuated on the physical robot or integrated for kinematic simulation. Some aspects of the IK solver will be described, and the choice of the state-of-art QP solver library *qpOASES* will be motivated. The work presented has been been developed as part of the robotic framework **OpenSoT**, developed at the Italian Institute of Technology [Roc+15a; Fan+15; Ajo+14; Lee+14]. The framework has been developed for application to our compliant humanoid robots, in particular to the COMAN [**tsagarakis12**] in a compliant inverse kinematics scheme, and WALK-MAN [**Tsagarakis:2016**] as part of the high-level control architecture used in the Darpa Robotics Challenge. The framework consists of a C++ library tailored for the control of hyper-redundant fixed/floating base robot. Experiments and results in the Darpa Robotics Challenge are presented later on in this section, while the software API and tools as will be presented in chapter 4.

3.1 OpenSoT: a library for Whole-Body Control of Humanoid Robots

From a very broad perspective, the inverse kinematics problem is that of mapping task-space objectives into joint-space commands. Typical techniques to map task-space commands to joint commands can be roughly classified as inverse kinematics or inverse dynamics schemes and can be implemented using a variety of low level controllers. **OpenSoT** is a novel whole body motion library developed at the Italian Institute of Technology (IIT), which belongs to the former group: our goal has been to develop a high performance and flexible library that can generate reliably complex and efficient whole body motions.

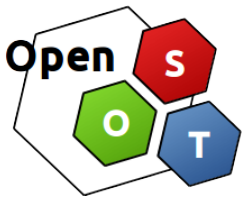


Fig. 3.1: OpenSoT Logo

The *Differential Inverse Kinematics Problem* consists of the determination of the joint trajectories corresponding to a given task for the end-effector in *Operational Space*. This problem is in general highly non-linear, a *closed form* solution may not be available, and may have multiple solutions. In the case in which the robot is *redundant* with respect to the task, that is, the degrees of freedom of the robot are greater than the degrees of freedom required by the specified task, the IK Problem admits an infinite number of solutions. In case of operational space control, at least a solution is guaranteed to exist if the given goal in operational space belongs to the manipulator *dexterous space*. Despite the relationship between joint position variables and operational space poses is not linear, the mapping between joint velocity variables and operational space velocities is linear through the robot Jacobian:

$$\dot{\mathbf{x}} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \tag{3.1}$$

where the Jacobian $\mathbf{J}$ (we will skip the dependency on the actual configuration $\mathbf{q}$ from now on) is expressed from a certain *base link* to a certain *distal link*, and operational space velocities $\dot{\mathbf{x}}$ of the *distal link* are expressed in the *base link* reference frame. A general solution, for the redundant case, of (3.1) is:

$$\dot{\mathbf{q}}_d = \mathbf{J}^\dagger \dot{\mathbf{x}}_d + \left(\mathbf{I} - \mathbf{J}^\dagger \mathbf{J}\right) \dot{\mathbf{q}}_0 \tag{3.2}$$

where $\mathbf{J}^\dagger$ is the *pseudo inverse* of the Jacobian and $\dot{\mathbf{q}}_0$ is an arbitrary joint space velocity. This solution allows also for *task prioritization*:

$$\dot{\mathbf{q}}_0 = \mathbf{J}_0^\dagger \dot{\mathbf{x}}_0 \tag{3.3}$$

in particular, the solution given by (3.2) with (3.3), projects the joint velocities of the second task into the null space of the first task removing all the components that would interfere with it [SK08]. Many other solutions of (3.1) are available in literature [SS91]. . Singularities in the Jacobian, as well as joint velocity limits, are often handled with the *Singularity Robust Inverse* (SRI) [Nak90], [MK89]. The

problem of this type of schemes is that they do not handle constraints in the form of inequalities.
To overcome this problem, techniques from the optimization field have been adopted. We can consider the inverse problem of (3.1) as a Quadratic Programming (QP) problem with linear constraints:

$$\begin{aligned} \dot{\mathbf{q}}_{\mathrm{d}} = \underset{\dot{\mathbf{q}}}{\operatorname{argmin}} \; & \|\mathbf{J}\dot{\mathbf{q}} - \dot{\mathbf{x}}_{\mathrm{d}}\| \\ s.t. \; & \mathbf{A}\dot{\mathbf{q}} \leq \mathbf{b} \end{aligned} \tag{3.4}$$

The two most popular families of methods to solve the problem in (3.4) are *active set* and *interior point* methods. *Active set* methods consider constraints as equalities, in the circumstance when they are active, so that the problem (3.4) is transformed in a QP with only equalities. At each iteration the new active set is computed checking for the infeasible constraints. *Interior point* methods use barrier functions (such as log functions) to penalize the cost function in the region where the bounds are violated and the solution is found by iteratively relaxing the barrier function. Even under the optimization point of view, the task prioritization can be handled considering the minimization of the cost function under equality constraints given by the minimization of previous tasks. This technique is commonly referred as *Stack of Tasks* (SoT) [Man+09].
OpenSoT implements the idea of decoupling atomic tasks/constraints descriptions and solvers to execute multiple tasks and achieve complex motion behaviors. It employs a solver implementing a cascade of QP problems, and a set of tasks and constraints in velocity space in order to solve a generic hierarchical inverse kinematics problem on a floating or fixed base robot. The stock IK solver consists on a state machine that hides all the complexity of the underneath QP solver based on a state-of-art library in QP resolution using the active set approach: *qpOASES* [Fer+13]. This yields the following features that make the implementation of **OpenSoT** unique and attractive:

- Demonstrates adequate modularity through the separation of task descriptions, control schemes and solvers maximizing customization, flexibility and expandability.

- Provides user friendly interfaces for defining tasks, constraints and solvers to promote integration and cooperation in the emerging field of whole-body hierarchical control schemes.

- Demonstrates computation efficiency to allow for real time performance implementations.

- Allows ease of use and application with arbitrary robots through the Universal and Semantic Robotic Description Formats (URDF and SRDF).

The reliability and robustness of **OpenSoT** framework has been proved during the DRC Finals since it has been used by the WALK-MAN team, to perform all the manipulation tasks involving *whole-body* motions while taking into account various type of constraints. During the development period, the **OpenSoT** framework allowed the fast evaluation of different formulations of the whole-boby control problem in order to optimally solve the tasks of the competition, thusly demonstrating the benefits of a flexible high-level framework in robotics rapid application development. The architecture of **OpenSoT** encourages collaboration and helps integration and code maintenance.

3.1.1 Related Work

Many approaches to solve the IK problem are presented in the literature and most of them address the singularity problem through Tikanov regularisation, with approaches similar to the *Singularity Robust Inverse* (SRI), but lack the possibility to specify constraints, whereas the aforementioned QP formulation allows for an elegant formulation of the IK problem considering linear constraints. **OpenSoT** is thus inspired by the *Stack of Tasks*, where the stock IK solver is based on QP (Quadratic Programming) optimization with the possibility to specify *hard* [Kan+09] and *soft* [Chi+91] priorities between tasks as well as linear constraints and bounds [Esc+14].

An interesting approach to hierarchical inverse kinematics resolution is given by [Liu+15] in the form of a Generalized Hierarchical Control (GHC). GHC is developed by means of a novel Generalized Projector which allows to project lower priority tasks into higher priority tasks fully or partially, thus implementing mixable *strict* and *non-strict hierarchies* (i.e. *hard* and *soft priorities*) using a single generalized projecting operation. This allows to mix strict and non-strict hierarchies and to smoothly change priorities, smoothly insert and remove tasks, as well as define complex priority relationships which are better described as a priority network rather than a lexicographic hierarchy.

In [Hau14] a method to optimize precomputed joint trajectories so as to satisfy contact and dynamic constraints is proposed.

In [DP+14] a way to specify hierarchical optimal problems is presented, with application to control of a humanoid robot.

A recent interesting approach is presented in [FDL14], where redundancy is resolved starting from the lowest priority task.

In [DS+07; Dec+09; Dec+13; Smi+09; Van+12; Van+13] the *iTaSC* framework is developed, which aims at providing a complete solution for specifying tasks and constraints in a generalized way and provide solvers to automatically compute the control law starting from the task specification. The framework includes also a Domain Specific Language (*DSL*) for rapid application development and formal validation. The method allows for different low-level solvers, and at the moment implements a weighted damped-least-squares pseudoinverse-based (*WDSL*-

psudoinverse) velocity control. To the authors knowledge, it has never been tested on humanoid robots.
In [ADS14] a task specification and solver is implemented using expression graphs, in a similar way (conceptually) to the Stack of Tasks.
In [Pat+10] a nonlinear optimization is also performed to perform IK on the iCub platform, where the focus is on Cartesian control of the end-effectors.
There are many other frameworks including those that are based on Inverse Dynamics, implemented on top of a pure low level torque control (a notable example is [Sen+10]), yet it is difficult to find hardware platforms mature enough to implement control schemes of such frameworks and up to now no complex tasks have been experimentally demonstrated yet in humanoid bipedal robots. Based on the operational space formalism the *Standard Control Library* [Men] implements the whole-body multi-task control framework previously cited and it is an updated version of [Phi+11]. A recent new library based on the same concept is [Fok+15].
In [**Beeson15**] their solution is compared to popular IK schemes as offered by *KDL* [Smi+11], highlighting the advantages of optimization approaches to the common classic IK schemes.
None of the considered libraries provides a complete set of implemented tasks, constraints and solvers that permits to describes and solve IK problems in an easy and versatile way.

3.1.2 A Framework for Whole-Body Control

Whole body control

Whole-body control here will be referred to those control laws that allow to execute tasks by using the whole robot structure. The whole-body control laws typically allow to execute several tasks at the same time when possible, by making use of the redundancies provided by the robot. Preferably they should be as generic as possible in allowing specification of tasks both in Operational and Cartesian space, and in particular, allow for the control in Operational space of any part of the robot, even when does do not coincide with the end-effectors. The whole-body control problem can be solved by many approaches, as shown in the previous section, but we will concentrate on a kinematic control law that allows to solve the whole-body control problem in generic kinematic trees, with particular emphasis to the generic case of a floating-base robot.

Inverse Kinematics

We consider a robot executing n tasks simultaneously, and for each of these tasks T_i, a proper error function $\mathbf{e}_i(\mathbf{q}, t)$ is provided, describing the task error. The time derivative of the error can be computed as

$$\dot{\mathbf{e}}_i = \frac{\partial \mathbf{e}_i}{\partial \mathbf{q}}\dot{\mathbf{q}} + \frac{\partial \mathbf{e}_i}{\partial t} = \mathbf{J}_i\dot{\mathbf{q}} + \frac{\partial \mathbf{e}_i}{\partial t} \tag{3.5}$$

with $\mathbf{J}_i$ the error Jacobian. During the execution of a generic task it is desirable that the task error converges to 0, by imposing an exponential dynamic, that is

$$\dot{\mathbf{e}}_i = -\lambda \mathbf{e}_i \Rightarrow \mathbf{J}_i \dot{\mathbf{q}} = -\frac{\partial \mathbf{e}_i}{\partial t} - \lambda \mathbf{e}_i = \dot{\mathbf{e}}_i^* \tag{3.6}$$

If the robot is redundant with respect to a task, secondary tasks can be also added and executed without affecting the performance of the primary task, and given a set of tasks described by the couple $T_i = (\mathbf{J}_i, \dot{\mathbf{e}}_i^*)$, the robot can be commanded to execute them using its whole body motion capabilities. In order to implement the method and obtain this result, the relative importance between tasks needs first to be defined. Thus, two kinds of relationships: *hard priority* and *soft priority* needs to be set. A task has a higher *hard priority* with respect to another task if the latter can not deteriorate the performance of the first one. *Soft priorities* are defined between tasks so all the solutions are influenced by each other proportionally to theirs weights, or in other terms, a tradeoff between the performance of competing task is obtained which is continuously tunable by means of a weight coefficient.

The execution of a hierachy of tasks related by an hard priority relation has a well-known solution in the stack of tasks, where hard priorities are enforced by the order of the task in the stack. To take into account also soft priorities the augmented Jacobian formulation [Chi+91] can be employed. It must be noted though that the augmented Jacobian formulation alone cannot enforce hard priorities since adding many tasks together can generate an ill-conditioned augmented Jacobian matrix. Therefore, to generate whole-body motions, a series of QP problems in cascade is instead solved [Kan+11]. This is a well known method to derive motions by executing tasks adding bilateral constrains to the inverse kinematics problem [Esc+14]. A generic task can be described as

$$\begin{aligned} \dot{\mathbf{q}}_i = \quad & \underset{\dot{\mathbf{q}}}{\text{argmin}} \quad \|\mathbf{J}_i \dot{\mathbf{q}} - \dot{\mathbf{e}}_i^*\| \\ & s.t. \quad \mathbf{A}_{c,1} \dot{\mathbf{q}} \leq \mathbf{b}_{c,i} \end{aligned} \tag{3.7}$$

The formulation used in (3.7) for the constraints can be profitably used to express lower and upper bounds for the variable value as well as equality constraints.

In general, the n^{th} task will then be written as

$$\begin{aligned} \dot{\mathbf{q}}_d = \quad & \underset{\dot{\mathbf{q}}}{\text{argmin}} \quad \left\| \mathbf{J}_n \dot{\mathbf{q}} + \lambda \mathbf{e}_n + \frac{\partial \mathbf{e}_n}{\partial t} \right\| \\ & s.t. \quad \mathbf{A}_1 \dot{\mathbf{q}} = \mathbf{A}_1 \dot{\mathbf{q}}_1 \\ & \qquad \vdots \\ & \mathbf{A}_{n-1} \dot{\mathbf{q}} = \mathbf{A}_{n-1} \dot{\mathbf{q}}_{n-1} \\ & \mathbf{A}_{c,1} \dot{\mathbf{q}} \leq \mathbf{b}_{c,1} \\ & \qquad \vdots \\ & \mathbf{A}_{c,n} \dot{\mathbf{q}} \leq \mathbf{b}_{c,n} \end{aligned} \tag{3.8}$$

where $\dot{\mathbf{q}}_d$ is the joint displacement (control variable) which minimizes the objective function and satisfy the given constraints, if they are a feasible set. In (3.8) the previous solutions $\dot{\mathbf{q}}_i,\ i < n$ are taken into account by specifying constraints of the type $\mathbf{A}_i\dot{\mathbf{q}} = \mathbf{A}_i\dot{\mathbf{q}}_i,\ \forall i < n$, so that the optimality of all higher priority tasks is not changed by the solving for the n-th task. While in (3.8) the first task has a relationship of hard priority with respect to the second, and so on, for each *level* of priority, a soft priority relationship between tasks can be imposed introducing the relative weights β_i, so that the augmented Jacobians and the error vectors can be written as

$$\begin{aligned}\mathbf{J}_{\text{aug}} &= \begin{bmatrix}\beta_1\mathbf{J}_1^T & \dots & \beta_n\mathbf{J}_n^T\end{bmatrix}^T \\ \mathbf{e}_{\text{aug}} &= \begin{bmatrix}\beta_1\mathbf{e}_1^T & \dots & \beta_n\mathbf{e}_n^T\end{bmatrix}^T\end{aligned} \tag{3.9}$$

where the soft priority between tasks is altered by tuning the relative weights β_i, with higher priority tasks having larger β_i. As already mentioned, in this case the tasks can still influence each other's performance. A mixture of hard and soft priorities is in general needed to describe a stack of tasks. The solution obtained can then be commanded directly to a velocity controlled robot or integrated to command a position controlled robot as

$$\mathbf{q}_d = \mathbf{q} + \dot{\mathbf{q}}\Delta t \tag{3.10}$$

where Δt is the control loop period.

OpenSoT

OpenSoT is a robotics library tool oriented to Whole-Body Control. The main idea behind its implementation is to decouple the task and constraint description to the solver used to solve the general Inverse Kinematics/Dynamics problem. **OpenSoT** aims at providing a standard way to describe the aforementioned entities in an atomic way, and at the same time to build a repository of common entities that can be reused to create a generic stack using a user friendly development and integration interface.

OpenSoT offers classes to describe different tasks and constraints whicah can be used to design the robot behavior during the task execution, and solvers with various capabilities and strategies to obtain solutions to the control problem. According to the control type, different controllers are available in literature. Defining a task once in a proper way allows to switch between different control laws, as shown in [Nak+08], thus decoupling task description and control type. The same applies to constraints and bounds [Fla+12]. Other than putting tasks in a stack to specify hard priorities during the solving phase, it is possible to perform a set of basic operations which allow composition (the *Aggregate* task), selection (the *SubTask* utility) and masking (the *ActiveJointMask* selector).

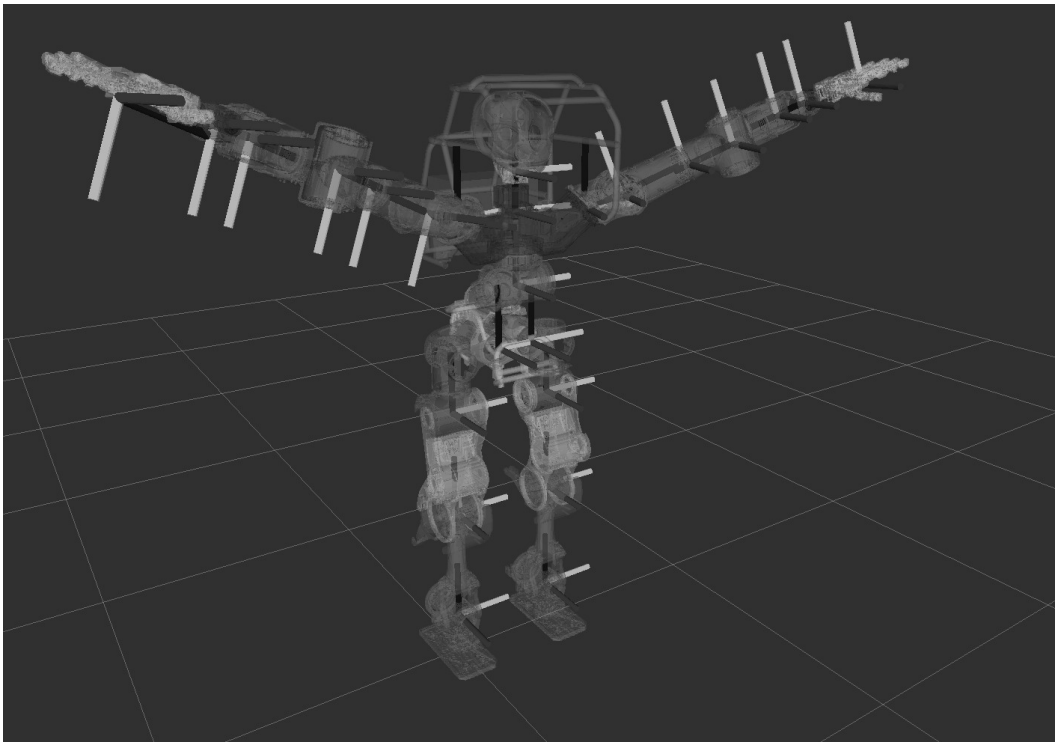


Fig. 3.2: WALK-MAN robot kinematics and reference frames

3.1.3 OpenSoT Tasks

In **OpenSoT**, a task T_i is in general defined as:

$$T_i = \mathcal{T}\left(\mathbf{J}_i,\, \dot{\mathbf{e}}_i^*\right) \tag{3.11}$$

For the solver, a task can be defined as a matrix and a vector, the task Jacobian and the task error respectively, which will get used by the solving algorithm. For the case of the QP stack,

$$T_i = \mathcal{T}\left(\mathbf{A}_i,\, \mathbf{b}_i\right) \tag{3.12}$$

where $\mathbf{A}_i^T\mathbf{A}_i$ is the task Hessian and $\mathbf{A}_i^T\mathbf{b}_i$ the task gradient, in this case expressed by choosing the identity metric for the QP problem. In general, a different metric can be expressed by correctly designing the W matrix the previous expression , which will coincide to a different metric during the minimizatin of the 2-norm of the residuals vector.

Task operations: the Math of Tasks

As previously specified, task operations can be used, together with the stacking operation, to build a complex stack out of simple stack definitions. These operations can be also defined using a simple semantic, which we call *Math of Tasks* (Table 3.1).

Aggregated The aggregated task constructs an augmented Jacobian starting from a definition of more basic tasks. This allows to enforce soft priorities between tasks

Operation	Symbol	Expression	Result
Stack	/	$S = T_1/T_2$	the stack S gets created, T_1 has higher priority
Augmentation	+	$T_3 = T_1 + T_2$	the augmented task T_3 is created, $n_3 = n_1 + n_2$
Applying Constraint	<<	$T_1 << C_0$	the constraint C_0 is attached to the task T_1

Tab. 3.1: Math of Tasks: Operations and respective symbols

that get incorporated into the augmented task Jacobian. The formulation can be written as

$$T_{\text{agg}} = \mathcal{T}\left(\left[\mathbf{J}_1^T \dots \mathbf{J}_n^T\right]^T, \quad \left[\mathbf{b}_1^T \dots \mathbf{b}_n^T\right]^T\right) \tag{3.13}$$

where $\mathbf{J}_i \in \mathbb{R}^{n_i \times n_{\text{dofs}}}$ The aggregated task represents a way to impose soft priorities between different tasks. The relative priorities can be tuned by selecting a proper weight matrix, $\mathbf{W}_{\text{agg}} \in \mathbb{R}^{n_{\text{agg}} \times n_{\text{agg}}}$ with $n_{\text{agg}} = \sum_i n_i$. Considering a diagonal weight matrix $\mathbf{W}$

$$\mathbf{W} = \begin{bmatrix} \beta_1 & & & \\ & \beta_2 & & 0 \\ & & \ddots & \\ & 0 & & \beta_{n-1} \\ & & & & \beta_n \end{bmatrix} \tag{3.14}$$

one would obtain the formulation from (3.9).

SubTask The *SubTask* allows to define tasks by selecting rows from a higher-dimensional task, $n_{\text{sub}} < n_{\text{task}}$. This allows to exert control only on the task dimensions of interest.

$$T_{\text{sub}} = \mathcal{T}\left(\left[\mathbf{J}_{r_1}^T \dots \mathbf{J}_{r_p}^T\right]^T, \quad \left[b_{r_1}^T \dots b_{r_p}^T\right]^T\right) \tag{3.15}$$

with

$$r_i \in [1, n_{\text{task}}], r_i \neq r_j$$

Where r_i is a row index and $\mathbf{J}_{r_i}$ is a row vector from the original task, b_{r_i} is the corresponding element from the vector $\mathbf{b}$. This will result in a subtask of size $p \leq n$.

ActiveJointMask The *Active Joint Mask* allows to effectively lock certain joints, meaning they will not get used by the specified task. It is implemented by substituting the columns of the joints to lock in the task Jacobian with a column of zeros,

$$T_{\text{masked}} = \left(\left[\mathbf{J}_{\text{masked},c_1} \dots \mathbf{J}_{\text{masked},c_n}\right], \quad \mathbf{b}\right) \tag{3.16}$$

where

$$\mathbf{J}_{\text{masked},c_i} = \begin{cases} \mathbf{J}_{c_i} & \text{joint is active} \\ 0_m & \text{joint is locked} \end{cases}$$

where $\mathbf{J}_{c_i}$ is the i-th column of the unmasked Jacobian, $0_m \in \mathbb{R}^{n_{\text{task}} \times 1}$.

Cartesian, CoM, MinimumCoMVelocity, MiminumCartesianVelocity It is possible to define a general Cartesian task as the *aggregate* of a Cartesian position task and a Cartesian orientation task. The Cartesian task computes the (relative) Jacobian between any given base and distal links in the kinematic tree, ${}^b\mathbf{J}_d$. The Cartesian errors in position and orientation are computed respectively as:

$$\begin{aligned} \mathbf{e}_p &= \mathbf{p}_d - \mathbf{p} \\ \mathbf{e}_o &= -(\eta_d \boldsymbol{\epsilon} - \eta \boldsymbol{\epsilon}_d + [\boldsymbol{\epsilon}_d \times] \boldsymbol{\epsilon}) \end{aligned} \tag{3.17}$$

and the task is defined as:

$$\begin{aligned} T_{C,p} &= \mathcal{T}\left({}^b\mathbf{J}_{d,p}, \quad \dot{\mathbf{p}}_d + \mathbf{K}_p \mathbf{e}_p\right) \\ T_{C,o} &= \mathcal{T}\left({}^b\mathbf{J}_{d,o}, \quad \boldsymbol{\omega}_d + \mathbf{K}_o \mathbf{e}_o\right) \end{aligned} \tag{3.18}$$

where $\mathbf{p}_d = [x_d \quad y_d \quad z_d]$ is the desired position and $\boldsymbol{\alpha}_d = [\eta_d \quad \epsilon_{1,d} \quad \epsilon_{2,d} \quad \epsilon_{3,d}]$ is the desired orientation expressed as a quaternion [Nak+08], $\mathbf{K}_p$ and $\mathbf{K}_o$ are positive definite matrices and $\boldsymbol{\xi}_d = [\dot{\boldsymbol{p}}_d \quad \boldsymbol{\omega}_d]$ is the desired Cartesian velocity for the end-effector. A particular case is the *CoM* task which is defined as a Cartesian position task. In both cases, the tasks can become minimum Cartesian velocity tasks by setting $\lambda = 0$.

Interaction The interaction task consists in force control through an admittance scheme. This task is built on top of the Cartesian task since pose and velocity references are generated from a wrench error:

$$\Delta\mathbf{x} = \mathbf{C}\left({}^{\text{b}}\mathbf{w}_{\text{d,ee}} - {}^{\text{b}}\mathbf{w}_{\text{m,ee}}\right) \tag{3.19}$$

and

$${}^{\text{b}}\mathbf{w}_{\text{m,ee}} = \begin{bmatrix} {}^{\text{b}}\mathbf{R}_{\text{ft}} & \mathbf{0} \\ \mathbf{0} & {}^{\text{b}}\mathbf{R}_{\text{ft}} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ {}^{\text{ft}}\mathbf{p}_{\text{ee}} \times & \mathbf{I} \end{bmatrix} {}^{\text{ft}}\mathbf{w}_{\text{m,ft}} \tag{3.20}$$

where ${}^{\text{b}}\mathbf{w}_{\text{d,ee}}$ is the desired wrench that the robot has to exert to the environment, ${}^{\text{ft}}\mathbf{w}_{\text{m,ft}}$ is the wrench that the robot is exerting to the environment measured on the force/torque sensor, ${}^{\text{b}}\mathbf{R}_{\text{ft}}$ is the rotation from the base link to the force/torque reference frame, ${}^{\text{ee}}\mathbf{p}_{\text{ft}}\times$ is the skew symmetric matrix computed from the distance between the force/torque reference frame and the end-effector reference frame, $\mathbf{C}$ is a positive definite compliance matrix. The computed $\Delta\mathbf{x}$ is used as feed-forward term and integrated as reference term for the Cartesian task.

MinimumEffort The minimum effort task is defined in joint space as:

$$T_{\text{mineffort}} = \mathcal{T}\left(\mathbf{I}, \quad \alpha_g \nabla(g(\mathbf{q})^T g(\mathbf{q}))\right) \tag{3.21}$$

where $g(\mathbf{q})$ are the torques due to gravity computed through Iverse Dynamics. The gradient is computed numerically by means of two-point estimation and the Hessian is set to be the identity, so as to effectively implement a *Gradient Projection Method* [Ros60].

Manipulability The manipulability task is again defined in joint space as:

$$T_{\text{manip}} = \mathcal{T}\left(\mathbf{I}, \quad -\alpha_g \nabla(h(\mathbf{q}))\right) \tag{3.22}$$

with

$$h(\mathbf{q}) = det\left(\sqrt{\mathbf{J}(\mathbf{q})^T \mathbf{J}(\mathbf{q})}\right) \tag{3.23}$$

being the manipulability index of the robot. As for the minimum effort task, the gradient is computed numerically. The minus sign is worth noticing since the task is a maximization of the chosen manipulability index of the robot.

MinimizeAcceleration The minimize acceleration task is also defined in joint space as:

$$T_{\text{minacc}} = (\mathbf{I}, \quad \lambda \dot{\mathbf{q}}_{\text{prev}}) \tag{3.24}$$

This task will try to minimize the change in velocity in joint space. When the λ of the task is equal to $\frac{1}{dT}$, it will become independent from the time step size and will effectively be a first order approximation of joint acceleration minimization.

Postural A postural task is defined in joint space as:

$$T_p = (\mathbf{I}, \quad \dot{\mathbf{q}}_d + \lambda(\mathbf{q}_d - \mathbf{q})) \tag{3.25}$$

The task can become a minimum velocity task by setting $\lambda = 0$ and $\dot{\mathbf{q}}_{\text{d}} = 0$, as in this case the cost function is reduced to $\|\dot{\mathbf{q}}\|_{\mathbf{W}}$.

Constraints and Bounds

Constraints model equalities and bilateral/unilateral inequalities. A generic unilateral constraint of the form

$$\mathbf{A}_{c,1}\dot{\mathbf{q}} \leq \mathbf{b}_{c,1} \tag{3.26}$$

can be expressed using the simple syntax:

$$\mathcal{C}\left(\mathbf{A}_c, \mathbf{b}_c\right) \tag{3.27}$$

Bilateral constraints and unilateral and bilateral bounds can be expressed in terms of unilateral constraints according to Table 3.2.

Constraint	Equation	Syntax	Equivalent Syntax
bilateral constraint	$\mathbf{b}_l \leq \mathbf{A}_c\dot{\mathbf{q}} \leq \mathbf{b}_u$	$\mathcal{C}_{\text{bilateral}}\left(\mathbf{A}_c,\, \mathbf{b}_{c,l},\, \mathbf{b}_{c,u}\right)$	$\mathcal{C}\left(\begin{bmatrix}\mathbf{A}_c^T & -\mathbf{A}_c^T\end{bmatrix}^T,\, \begin{bmatrix}\mathbf{b}_{c,u}^T\, \mathbf{b}_{c,l}^T\end{bmatrix}^T\right)$
unilateral bounds	$\dot{\mathbf{q}} \leq \mathbf{b}_{c,u}$	$\mathcal{B}\left(\mathbf{b}_{c,u}\right)$	$\mathcal{C}\left(\mathbf{I}_{n_{\text{dofs}}},\, \mathbf{b}_{c,u}^T\right)$
bilateral bounds	$\mathbf{b}_l \leq \dot{\mathbf{q}} \leq \mathbf{b}_u$	$\mathcal{B}_{\text{bilateral}}\left(\mathbf{b}_{c,l},\, \mathbf{b}_{c,u}\right)$	$\mathcal{C}\left(\begin{bmatrix}\mathbf{I}_{n_{\text{dofs}}} & -\mathbf{I}_{n_{\text{dofs}}}\end{bmatrix}^T,\, \begin{bmatrix}\mathbf{b}_{c,u}^T\, \mathbf{b}_{c,l}^T\end{bmatrix}^T\right)$

Tab. 3.2: Constraints and bounds. Transforming unilateral constraints and bounds from upper bounds to lower bounds is trivial and therefore not included in the table

Constraints and bounds can be applied to a single task (i.e. $T_i << \mathcal{C}$) or to the whole stack (i.e. $S << \mathcal{C}$): applying constraints to the stack is equivalent to applying them to every task in the stack. We will call the former *local* constraints, while we will refer to the latter as *global*. In general, when a *local* constraint is applied to a certain task, it should be applied also to all its lower priority tasks in the stack to make sure that it will be enforced in the final solution. There are, anyway, conditions when a constraint does not need to be enforced directly to lower priority tasks, since its enforcement is implicit in keeping optimality of the higher priority tasks. Consider a task $T_i = \mathcal{T}\left(J_i, \dot{\mathbf{e}}_i^*\right)$, $\mathbf{J} \in \mathbb{R}^{m\times n}$ with $rank(\mathbf{J}) = m$, a stack S composed of n tasks, and a constraint $C_j = \mathcal{C}\left(\mathbf{A}_{c,j},\, \mathbf{b}_{c,j}\right)$. If we apply the constraint to the task $T_i << C_j$, if

$$rank\left(\begin{bmatrix}\mathbf{J}_i \\ \mathbf{A}_j\end{bmatrix}\right) = m \tag{3.28}$$

then C_j can be applied as a local constraint to T_i, and it will automatically be enforced on lower priority tasks up to the final solution $\dot{\mathbf{q}}_n^* = \dot{\mathbf{q}}_d$ $(T_j \forall j > i)$ since it will be implicit in the solution $\dot{\mathbf{q}}_i$.

Aggregated The aggregated constraint performs merging of a list of constraints into a single constraint, in particular piling the equalities and inequalities constraints, and merging the bounds as follows:

$$C_{\text{agg}} = \mathcal{C}\left(\begin{bmatrix}\mathbf{A}_{c,1}^T & \mathbf{A}_{c,2}^T\end{bmatrix}^T,\quad \begin{bmatrix}\mathbf{b}_{c,1}^T & \mathbf{b}_{c,2}^T\end{bmatrix}^T\right) \tag{3.29}$$

$$b_{\text{agg}} = \left(\max\left(\mathbf{l}_1, \mathbf{l}_2\right),\quad \min\left(\mathbf{u}_1, \mathbf{u}_2\right)\right) \tag{3.30}$$

ConvexHull The CoM is bounded to lay inside the convex hull defined by the contacts with the environment (Figure 3.3), where we can write

$$C_{\text{CH}} = \mathcal{C}\left(\begin{bmatrix} a_0 & b_0 \\ \vdots & \vdots \\ a_{n-1} & b_{n-1}\end{bmatrix},\quad \begin{bmatrix} -c_0 \\ \vdots \\ -c_{n-1}\end{bmatrix}\right) = \mathcal{C}\left(\mathbf{A}_{\text{CH}},\, \mathbf{b}_{\text{CH}}\right) \tag{3.31}$$

with a_i,b_i,c_i coefficients of the implicit equation of the line $a_i x + b_i y + c_i = 0$, bounding the convex hull. These lines are expressed in a frame attached to the

CoM and parallel to the inertial frame, and are obtained by finding the coefficients of a line passing through two consecutive points of the convex hull of the support polygon. The convex hull is obtained by creating a hull of a point cloud of contact points between the foot and the ground, which can be obtained by skin sensors or, in current implementation, by the foot model assuming full-foot contact with the ground. In order to keep the vector $\mathbf{b}_{\text{CH}}$ expressed in the same units in which the

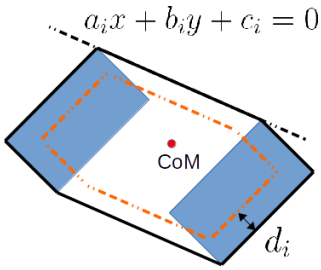


Fig. 3.3: The convex hull constraint for the CoM

points in the Convex Hull are expressed with respect to a certain frame, then the convex hull lines coefficients are normalized, so that for the i'th line, the coefficients will be scaled by the factor $d_i = \sqrt{a_i^2 + b_i^2}$.

BilateralConstraint This constraint is a generic constraint that can be built on the fly manually specifying a constraint matrix, $\mathbf{A}_{\text{user}}$, and upper and lower bounds $\mathbf{b}_{l,\text{user}}$, $\mathbf{b}_{u,\text{user}}$

$$C_{\text{bc}} = \mathcal{C}_{\text{bilateral}}\left(\mathbf{A}_{\text{user}},\, \mathbf{b}_{l,\text{user}},\, \mathbf{b}_{u,\text{user}}\right) \tag{3.32}$$

CartesianPositionConstraint The constraint implements limits on the Cartesian position of any feature on the robot body with respect to a defined hull in Cartesian space.

CoMVelocity This constraint implements limits on the Cartesian velocity of the *CoM* w.r.t. another link, or the inertial frame

$$C_{\text{C,p}} = \left({}^{l}\mathbf{J}_{CoM},\quad {}^{l}\dot{\mathbf{x}}_{\text{max,CoM}}\right) \tag{3.33}$$

TaskToConstraint This constraint allows to transform any task into a constraint. It is applicable for the cases when a certain task can be performed with zero error, e.g. the loop closure equation between the two feet of a humanoid robot in a double support stance.

JointLimits The joints limit constraint is one of the fundamental constraints to use in the real robot. It prevents to hit the joint limits reaching them at zero speed:

$$b_{\text{JointLimits}} = \left(\sigma\left(\mathbf{q}_{\text{min}} - \mathbf{q}\right),\quad \sigma\left(\mathbf{q}_{\text{max}} - \mathbf{q}\right)\right) \tag{3.34}$$

with $\left(\mathbf{q}_{\text{min}},\ \mathbf{q}_{\text{max}}\right)$ the joint limits.

JointVelocity Joint velocity limits permits to generate trajectories with bounded velocities:

$$b_{\text{JointVelocity}} = (-\sigma\dot{\mathbf{q}}_{\max}\Delta t, \quad \sigma\dot{\mathbf{q}}_{\max}\Delta t) \tag{3.35}$$

with $\dot{\mathbf{q}}_{\max}$ the maximum allowed joint velocity.

Dynamic Filter One of the fundamental problems in IK is that some assigned Cartesian reference trajectories might be dynamically unfeasible by the robot. This means that the robot might get damaged since the required joint torque for a certain motion could be too high. Various techniques have been presented in the past to avoid this problem, one of the most famous is the *Dynamic Filter* [Yam04]. This technique basically uses an ID step to filter the generated joint accelerations from the IK solution.
The *Dynamic Filter* can be formulated as a constraint. The dynamics of the robot can be written as:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) = \boldsymbol{\tau} - \mathbf{J}_c^T\mathbf{f}_c \tag{3.36}$$

where $\mathbf{M}(\mathbf{q})$ is the joint space inertia matrix, $\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$ takes into account centrifugal and Coriolis terms, $\mathbf{G}(\mathbf{q})$ are the gravity torques, $\boldsymbol{\tau}$ are the joint torques and $\mathbf{J}_c^T\mathbf{f}_c$ are the torques due to contacts (that we measure on the force/torque sensors). Considering an acceleration level control and taking into account that each joint can provide $[\tau_{\text{i,min}},\ \tau_{\text{i,max}}]$, it is possible to write the constraint as:

$$\mathbf{D}(\mathbf{q},\dot{\mathbf{q}}) + \boldsymbol{\tau}_{\min} \leq \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} \leq \mathbf{D}(\mathbf{q},\dot{\mathbf{q}}) + \boldsymbol{\tau}_{\max} \tag{3.37}$$

with $\mathbf{D}(\mathbf{q},\dot{\mathbf{q}},\mathbf{f}_c) = -\left(\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) - \mathbf{J}_c^T\mathbf{f}_c\right)$. In this work we are considering velocity level control, so it is possible to approximate the joint acceleration $\ddot{\mathbf{q}}$ as:

$$\ddot{\mathbf{q}} = \frac{\dot{\mathbf{q}} - \dot{\mathbf{q}}_{\text{prev}}}{\Delta T} \tag{3.38}$$

then the constraint can be rewritten at the velocity level as:

$$\Delta T\left(\mathbf{D}(\mathbf{q},\dot{\mathbf{q}}) + \boldsymbol{\tau}_{\min}\right) + \mathbf{M}(\mathbf{q})\dot{\mathbf{q}}_{\text{prev}} \leq \mathbf{M}(\mathbf{q})\dot{\mathbf{q}} \leq \Delta T\left(\mathbf{D}(\mathbf{q},\dot{\mathbf{q}}) + \boldsymbol{\tau}_{\max}\right) + \mathbf{M}(\mathbf{q})\dot{\mathbf{q}}_{\text{prev}} \tag{3.39}$$

A similar idea was presented also in [Par+98] but contact forces were not taken in consideration. Practically speaking, as we did for other constraints, it is useful to have a scaling factor $\sigma \in (0,\ 1]$ in front of the constraint:

$$\sigma\left(\Delta T\left(\mathbf{D}(\mathbf{q},\dot{\mathbf{q}}) + \boldsymbol{\tau}_{\min}\right) + \mathbf{M}(\mathbf{q})\dot{\mathbf{q}}_{\text{prev}}\right) \leq \mathbf{M}(\mathbf{q})\dot{\mathbf{q}} \leq \sigma\left(\Delta T\left(\mathbf{D}(\mathbf{q},\dot{\mathbf{q}}) + \boldsymbol{\tau}_{\max}\right) + \mathbf{M}(\mathbf{q})\dot{\mathbf{q}}_{\text{prev}}\right) \tag{3.40}$$

Self-Collision Avoidance Constraint

In our framework, the self-collision avoidance constraint will be formulated as an inequality constraint for each task in the IK Problem to guarantee the safety of the

robot. The basic idea is to calculate the minimum distance between every link pair in every control loop and obtain a list of several closest link pairs according to these minimum distances. For each of these link pairs of interest, we try to control the relative velocity of the one link with respect to the other in the direction connecting the closest points on the two links respectively, which is called velocity damping firstly introduced in [FT87] and reformulated inside the SoT [Kan+08], to avoid the potential collision between them.

So, to accelerate the computation of minimum distance of each link pair, we should employ some simpler collision model for each link first. In our paper, the capsule, sphere-swept line, is chosen among a lot of types of bounding volumes because of the convenience of the computation. As shown in Figure 3.4, the minimum distance of the link pair can be just considered as the minimum distance between the inner line segments minus the sum of the radii of the capsules for the two links respectively. The method proposed in [EK+13] is employed to generate the minimum bounding capsule for each of exact link body geometries represented by mesh. Furthermore, the minimum distance between a capsule pair and the shortest points on them can be easily obtained by using the Flexible Collision Library (FCL) [Pan+12]. FCL is a fully templated library which aims at general proximity calculation and collision detection on various types of collision geometries such as axis-aligned bounding box (AABB) and oriented bounding box (OBB). As of version 3.0 of the library, the capsule collision geometry has been added [Kne14].

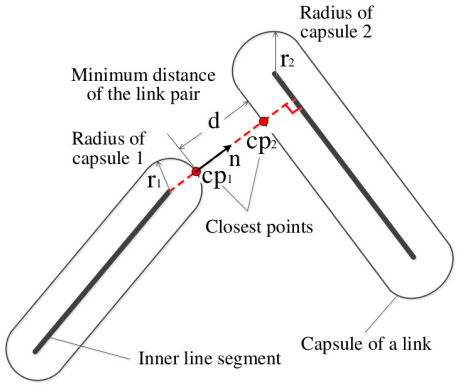


Fig. 3.4: Schematic diagram of the collision avoidance constraint on a capsule pair representing the collision model for a link pair

Once the closest points are computed, for instance, $\boldsymbol{cp}_1$ and $\boldsymbol{cp}_2$ shown in Figure 3.4, the relative motion of the capsule pair is going to be restricted in the direction:

$$\boldsymbol{n} = \frac{\boldsymbol{cp}_2 - \boldsymbol{cp}_1}{\|\boldsymbol{cp}_2 - \boldsymbol{cp}_1\|} \tag{3.41}$$

Where the distance $\|\boldsymbol{cp}_2 - \boldsymbol{cp}_1\|$ is calculated according to the Euclidean L2-norm. So, the relative velocity constraint in this direction can be formulated as follows:

$$\boldsymbol{n}^T[\boldsymbol{J}(\boldsymbol{cp}_1, \boldsymbol{q}) - \boldsymbol{J}(\boldsymbol{cp}_2, \boldsymbol{q})]\dot{\boldsymbol{q}} \leq \varepsilon \frac{d - d_s}{\Delta t} \tag{3.42}$$

In (3.42), $\boldsymbol{J}(\boldsymbol{cp}_1, \boldsymbol{q})$ and $\boldsymbol{J}(\boldsymbol{cp}_2, \boldsymbol{q})$ refer to the Jacobian matrices of the frames located at the closest points $\boldsymbol{cp}_1$ and $\boldsymbol{cp}_2$ respectively with respect to the base frame. Since the relative location of the capsule relative to the corresponding link frame is fixed, these two Jacobians can be obtained based on the Jacobians of their link frames by some simple transformation. It is worth noting that only the linear velocity component of the Jacobian (the first three rows) is taken into account in this case. Then, the formula $[\boldsymbol{J}(\boldsymbol{cp}_1, \boldsymbol{q}) - \boldsymbol{J}(\boldsymbol{cp}_2, \boldsymbol{q})]\dot{\boldsymbol{q}}$ can be considered as the relative velocity of the capsule pair. In addition, d_s is the safety distance which is the shortest distance allowed for the capsule pair, Δt denotes the control loop period, and ε indicates the gain value which is used to smoothly reduce the relative approaching velocity at which the threshold distance is reached. So, the inequality in (3.42) means the velocity projected from the original relative velocity of the capsule pair onto the direction connecting the closest points on the capsules would never make the distance between them smaller than the safety distance d_s, which is then used to avoid potential collision between the corresponding link pair. Furthermore, for multiple link pairs, the constraint formulation evolves to:

$$\begin{aligned}
\begin{pmatrix} \boldsymbol{n}_1{}^T \bar{\boldsymbol{J}}_1 \\ \vdots \\ \boldsymbol{n}_k{}^T \bar{\boldsymbol{J}}_k \end{pmatrix} \dot{\boldsymbol{q}} &\leq \begin{pmatrix} \bar{d}_1 \\ \vdots \\ \bar{d}_k \end{pmatrix} \\
\boldsymbol{N}\,\dot{\boldsymbol{q}} &\leq \boldsymbol{D} \\
\text{where} \quad \bar{\boldsymbol{J}}_i &= \boldsymbol{J}(\boldsymbol{cp}_{1,i}, \boldsymbol{q}) - \boldsymbol{J}(\boldsymbol{cp}_{2,i}, \boldsymbol{q}) \\
\text{and} \quad \bar{d}_i &= \varepsilon_i \frac{d_i - d_{s,i}}{\Delta t}
\end{aligned} \tag{3.43}$$

In this case, in order to add the self-collision avoidance constraint for k pairs of links in the IK Problem, the inequality constraint of every task should be replaced by $\boldsymbol{N}\dot{\boldsymbol{q}} \leq \boldsymbol{D}$ in (3.43).

3.1.4 Robust IK Solver

OpenSoT provides a class to implement different *Solvers* that use the classes *Tasks*, *Constraints* and *Bounds* to solve the IK problem for a desired control type. As mentioned before, the IK is a fundamental part in the control scheme as it maps the desired references in the operational space to desired references in joint space. This is a potentially dangerous step for many reasons that depend largely on the algorithm chosen to solve the IK problem. For this reasons an IK solver must be *robust* and we define two main characteristics: *singularity robustness* and *constraints/bounds handling*. The first one is required anytime the robot is near the kinematics singulari-

ties and the algorithm has to assure that no high joint velocities are generated. The latter is fundamental to prevent that a task potentially damages the robot.
Considering these requirements, we developed the standard IK solver provided with **OpenSoT** based on a sequence of QP optimizations with the possibility to specify *hard* and *soft* priorities between tasks as well as linear constraints and bounds. The solver is based on the framework (3.8):

$$\begin{aligned} \underset{\dot{\mathbf{q}}}{\operatorname{argmin}} \; & \|\mathbf{J}_i\dot{\mathbf{q}}_i - \mathbf{v}_{d,i}\|_{\mathbf{W}} + \lambda\|\dot{\mathbf{q}}_i\| \\ s.t. \; & \mathbf{c}_{\mathrm{l,i}} \leq \mathbf{A}_i\dot{\mathbf{q}}_i \leq \mathbf{b}_{\mathrm{u,i}} \\ & \mathbf{b}_{\mathrm{l}} \leq \mathbf{A}\dot{\mathbf{q}}_i \leq \mathbf{b}_{\mathrm{u}} \\ & \mathbf{u}_{\mathrm{l}} \leq \dot{\mathbf{q}}_i \leq \mathbf{u}_{\mathrm{u}} \\ & \mathbf{J}_{i-1}\dot{\mathbf{q}}_{i-1} = \mathbf{J}_{i-1}\dot{\mathbf{q}}_i \\ & \vdots \\ & \mathbf{J}_0\dot{\mathbf{q}}_0 = \mathbf{J}_0\dot{\mathbf{q}}_i \end{aligned} \tag{3.44}$$

where $\mathbf{J}_i$ and $\mathbf{v}_{d,i}$ are respectively the Jacobian and the desired velocity reference for i-th task, λ is a weight for the damping correction, $\mathbf{A}_i$, $\mathbf{c}_{\mathrm{l,i}}$ and $\mathbf{c}_{\mathrm{u,i}}$ are constraints present only in the i-th task, $\mathbf{A}$, $\mathbf{b}_{\mathrm{l}}$ and $\mathbf{b}_{\mathrm{u}}$ are global constrains as well as $\mathbf{u}_{\mathrm{l}}$ and $\mathbf{u}_{\mathrm{u}}$ are bounds that are present in all the tasks. The final set of constraints represent the optimality conditions that comes from the higher priority level tasks in the stack. In our IK solver all the optimality conditions are set automatically given the IK Problem while the other constraints are set by the user. As shown in [Nak90] the second term in the cost function guarantees the robustness near kinematics singularities, the λ gain can be chosen according to different criteria). Bounds and constraints are mandatory in order to be robust to joint limits and joint velocity/acceleration/torque limits. Conceptually, the IK solver is divided in two main parts: the *front-end* and the *back-end*. The *front-end* takes an IK Problem and prepare all the QP Problems that needs to be solved considering priorities, local and global constraints and bounds. The *back-end* consists of the underneath QP Solver that solves the single QP Problem. Tasks, constraints and bounds are defined as *pointers* and updated outside the solver at every step.

Back-End

As mentioned before, the *back-end* is based on a popular QP solver library called *qpOASES* [Fer+13] which implements an active-set approach to handle inequality constraints. The library provides also a *warm-start* and a *hot-start* approach to solve the QP Problem. Basically, in the *warm-start*, an initial guess from the previous solution and previous active set is used. This permits to speed up the computation with respect to a classical solution (that we call *initialization* because it has to be performed at least in the beginning). In the *hot-start* also the previous decomposition of the matrix for the KKT conditions is used to speed up computation.

When a QP has to be solved, the *back-end* first tries to solve the problem using the *hot-start*, if an error occurs then the *warm-start* is used. If also the *warm-start* fails a new *initialization* of the QP Problem is performed. If also the *initialization* fails then the solver will not generate a solution and notify the user by returning a *false* value.

Front-End

The *front-end* prepares all the optimization problems considering the presence of local/global constraints, bounds and priorities. For each task, the cost function is computed as:

$$f(\mathbf{q}_i) = \dot{\mathbf{q}}_i^T \mathbf{J}_i^T \mathbf{W}_i \mathbf{J}_i \dot{\mathbf{q}} + 2(\mathbf{J}_i \dot{\mathbf{q}})^T \mathbf{W} \mathbf{v}_{\mathrm{d,i}} \tag{3.45}$$

where the second term in (3.44) is automatically added by the solver so the user has only to set the λ value. If local constraints are present in the task, they are added to the matrix of the constraints together with the global ones. If the task is not the one at highest priority, the optimality constraints are computed as:

$$\mathbf{J}_j \dot{\mathbf{q}}_j = \mathbf{J}_j \dot{\mathbf{q}}_i \tag{3.46}$$

where $j = 0, 1, ..., i-1$ and $\dot{\mathbf{q}}_j$ are the previous computed solutions. The optimality constraints are added together with the other constraints automatically by the *front-end*. Equalities and inequalities constraints are treated together. When the preparation is finished, the solve method of the *back-end* is called (if the QP Problem has been initialized before, otherwise the initialization method is called).

Through the *front-end* interface, the user can set parameters regarding each problem in the *back-end*. It is possible to set different options to configure *qpOASES* to favor speed instead of reliability and vice versa. The *front-end* takes care to set default options for *qpOASES*.

Considerations on the IK Problem

It is worth noticing that not only the IK solver but also the IK problem will affect the resulting joint trajectory given by the high-level task solution. When writing the IK problem, the user may specify different gains and parameters as well as constraints and stacks that will influence in some way the result. From our experience, we used different ways to affect the solution: tuning the λ and setting gains for the Cartesian tasks in the cost function of (3.44), add constraints and bounds (however it may result in an unfeasible problem), add a joint space task at the lowest priority level. In the latter case, is well known that when the joint space task is the minimization of the joint velocity, the resulting optimization is equivalent to a weighted pseudo inverse [Sic+08].

3.1.5 Experimental Validation

In this section a large set of tests and experiments will be presented to evaluate the performances of the **OpenSoT** library and to show how the framework can be used for many different high-level tasks such as walking, manipulation and interaction with the environment. We also consider different robots (simulated and real hardware) that intend to demonstrate how the library is robot agnostic.

Soft Interaction and Tele-operation

[Set+14] For these experiments the *Compliant Inverse Kinematics* scheme has been used as shown in Figure 3.5. The IK block (**OpenSoT**) generates joint position refer-

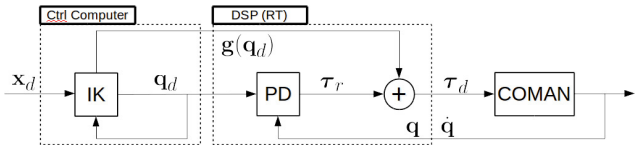


Fig. 3.5: Block diagram for the soft-interactive controller

ences and torques for gravity compensation for the second block that consists in a joint impedance controller. In particular the joint impedance control is implemented in a decentralized way at each DSP in the COMAN robot. The desired torque sent to actuator i is locally computed as

$$\tau_{d,i} = k_{q,i}(q_{d,i} - q_i) - k_{d,i}\dot{q}_i + g(q_d)_i \tag{3.47}$$

where q_i and $q_{d,i}$ are respectively actual and desired joint positions, $\dot{q}_i$ is the actual joint velocity, $k_{q,i}$ is a positive joint stiffness, $k_{d,i}$ is a positive joint damping and $g(q_d)_i$ is a gravity compensation torque computed at the desired joints configuration. We consider the following IK problem that is particularly suited for manipulation tasks:

$$\left(\begin{array}{c}\left(T_{\substack{\text{Right}\\\text{Wrist}}} + T_{\substack{\text{Left}\\\text{Wrist}}} + T_{\text{CoM}} + T_{\substack{\text{Right}\\\text{Foot}}}\right) << \left(C_{\substack{\text{CoM Velocity}\\\text{Limit}}} + C_{\substack{\text{CoM}\\\text{ConvexHull}}}\right) \backslash \\ \left(T_{\substack{\text{Joint}\\\text{Posture}}} + T_{\substack{\text{Minimum}\\\text{Effort}}}\right)\end{array}\right) << \left(B_{\substack{\text{Joint}\\\text{Limits}}} + B_{\substack{\text{Joint Velocity}\\\text{Limits}}}\right) \tag{3.48}$$

The gain for the Minimum Effort task is $(1 - \beta)$ while for the Joint Posture task is β. The parameter $\beta \in [0,\ 1]$ allows to smoothly on-line weight between a *postural* joint space task to *minimum effort* joint space task. Figure 3.6 presents the 2-norm of the torques in all the 29 joints of the robot. It is easy to recognize the different part where the last task is pure postural ($\beta = 1.0$) and where the last task is pure min effort ($\beta = 0.0$). The difference between the 2-norm of the torques in pure postural task and in pure min effort task is around $2[Nm]$ for the given references. The second figure in Figure 3.7 depicts the 2-norm of CoM position error. The error

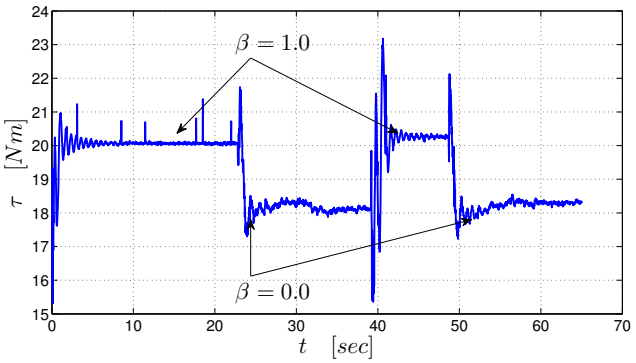


Fig. 3.6: 2-norm of torques during switching from a pure postural task to a mix of postural and minimum effort task

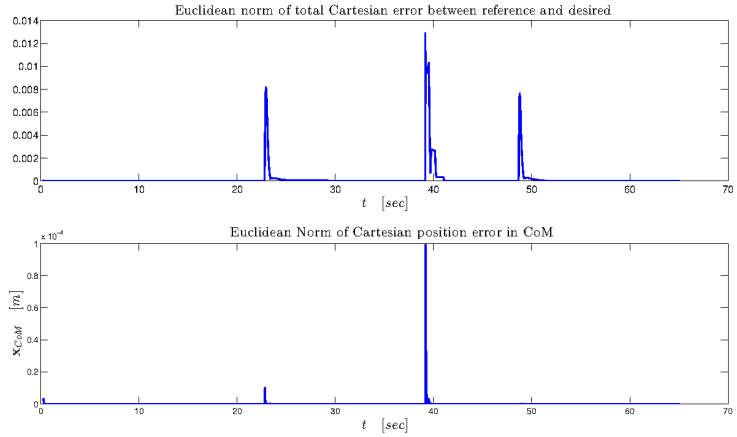


Fig. 3.7: 2-norm of Cartesian errors for CoM and end effectors

for a certain kinematic chain is computed as the sum of the 2-norms of the Cartesian error between the desired and computed Cartesian pose. It can been seen that the CoM is adequately maintained with the predefined reference position even during the switching of the tasks. Error in joint trajectory generation increases only during switching phases. The 2-norm of the vector of the joint errors is presented in the first figure in Figure 3.7. Finally in Figure 3.8 the COMAN is shown in different postures arising by changing β in the secondary task, the COMAN kinematic structure consists in 6 DoFs legs, 7 DoFs arms and a 3 DoFs torso. This particular IK problem and low-level controller have been also used to perform tasks with high level of interaction with the environment such as a writing task, Figure 3.10, a cleaning task, Figure 3.9 and a handover task. The joint impedance control makes the interaction possible and safe reducing impact/contact forces that may compromise the robot stability, damage the robot itself or the environment. The whole-body IK runs in open loop to prevent the action of the feedback during interaction from counteracting the effect of compliant behavior.

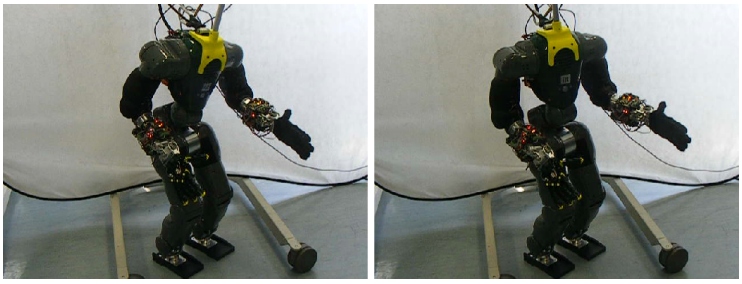


Fig. 3.8: COMAN holding end-effectors pose while executing a postural task (left) or a minimum effort task (right)

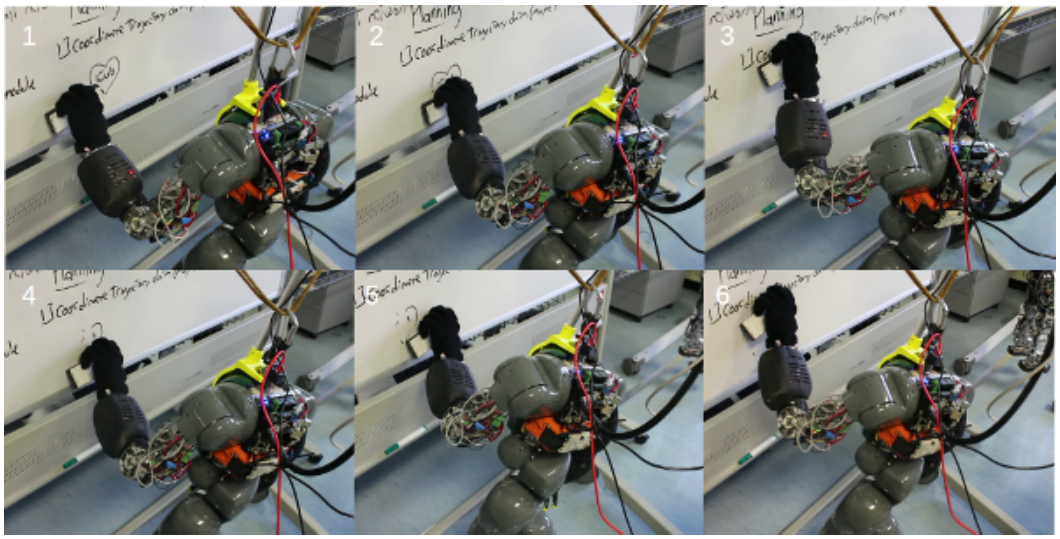


Fig. 3.9: COMAN erasing a whiteboard, interaction is handled by the joint impedance control

Fig. 3.10: COMAN performing a drawing task on a desk, interaction is handled by joint impedance control

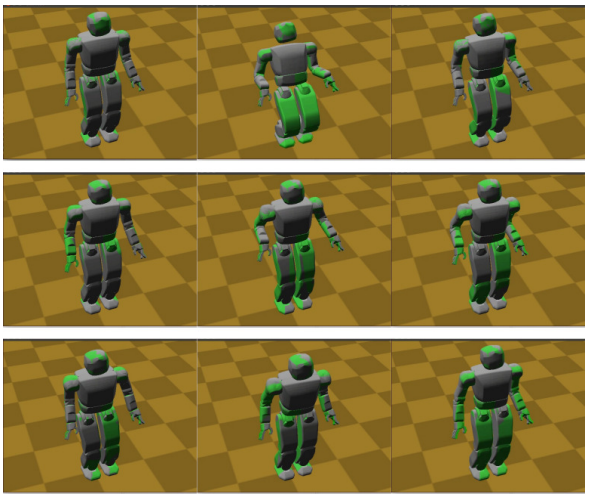


Fig. 3.11: HUBO performing whole-body tasks: from top to bottom, squat, arm trajectories (circle on left wrist, up-down with right wrist) and circular CoM trajectories on the x-y plane respectively

Examples

In this part we focus on a set of tasks and constraints that have been used in practice in a series of stacks of practical interest. The experiments show their behaviour and performances.

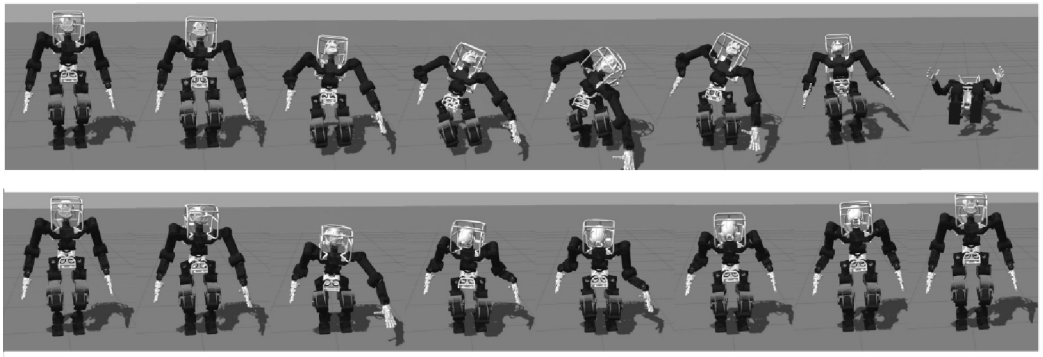


Fig. 3.12: In the upper sequence WALK-MAN falls due to a dynamically unfeasible motion while in the lower sequence the motion is dynamically feasible thanks to the dynamics constraint

Dynamic Filter As explained before, the idea behind the *Dynamic Filter* is to use the dynamics of the robot to filter non-feasible motions. In this experiment the effect of the dynamics as a constraint, at the velocity level, is shown. The robot has to perform a squat motion that requires a certain amount of torque on the legs. The same motion is then performed with the constraints considering that the available torque at the legs is 60% less ($20\ [Nm]$ instead of $50\ [Nm]$). The IK Problems considered is:

$$\begin{pmatrix} T_{\substack{\text{Right}\\\text{Foot}}} \backslash \\ \left(T_{\substack{\text{Right}\\\text{Wrist}}} + T_{\substack{\text{Left}\\\text{Wrist}}} + T_{\text{Waist}} + T_{\text{Torso}} + T_{\text{CoM}} \right) \backslash \\ T_{\substack{\text{Joint}\\\text{Posture}}} \end{pmatrix} << \left(B_{\substack{\text{Joint}\\\text{Limits}}} + B_{\substack{\text{Joint Velocity}\\\text{Limits}}} + C_{\text{Dyn}} \right) \tag{3.49}$$

The application of the robot dynamics Constraint into a complex IK problem to perform a Cartesian task while squatting with the simulated model of the humanoid robot WALK-MAN is showed in Figure 3.12. The task consists of moving the left arm forward and near the ground generating a squat motion of the whole body and high joint torques. We will show not only that the joint torques are bounded in the limits but also that the task makes the robot fall if performed without the robot dynamics constraint.

Apart from the robot dynamics constraint, we consider joint limits, joint velocity limits (up to $0.9\ \left[\frac{rad}{sec}\right]$), the Center of Mass of the robot has to lie inside the support

polygon and self collision avoidance. The self collision avoidance constraint makes use of capsules as bounding volumes to approximate the link geometries of the robot, in order to compute the minimum distance between colliding links.

For the robot dynamics constraint we are using $\sigma = 0.2$ and we are filtering the sensed (simulated) wrenches at the force/torque sensors using a simple filter:

$$\mathbf{w}_t \mathrel{+}= (\mathbf{w}_t - \mathbf{w}_{t\text{-}1})\, 0.6 \tag{3.50}$$

Furthermore we want to limit the amount of torque used in the torso to perform the task: in particular, for the three joints in the torso we set a maximum of $72\,[Nm]$ for the roll joint, $132\,[Nm]$ for the pitch joint and $72\,[Nm]$ for the yaw joint (around 40% less than the maximum available peak torques in the real robot).

The Cartesian task consists of a linear trajectory for the left hand, from the initial pose, $0.7\,[m]$ forward, $0.08\,[m]$ on the left, $0.5\,[m]$ down and it has to rotate around the z axis of $\frac{\pi}{3}\,[rad]$ and then back again. The trajectory has to be executed in 6 seconds.

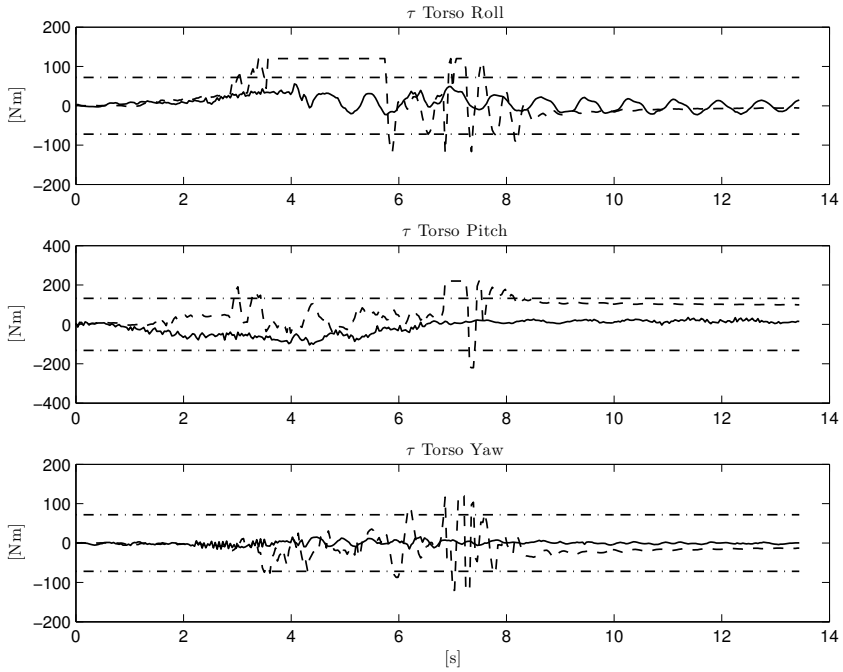


Fig. 3.13: Measured torques on the joints of the torso while performing the task without (dashed lines) and with (continuous lines) the robot dynamics constraint. The constant lines shows the limits on the torques

In Fig. 3.12 it can be observed the final motion performed by the robot when the robot dynamics constraint is not active (upper sequence) and when it is active (lower sequence). Without considering torque limits the robot falls in the second part of the squat motion. Fig. 3.13 shows that the torques at the torso remains in the limits when using the robot dynamics constraint, while saturate when not using it.

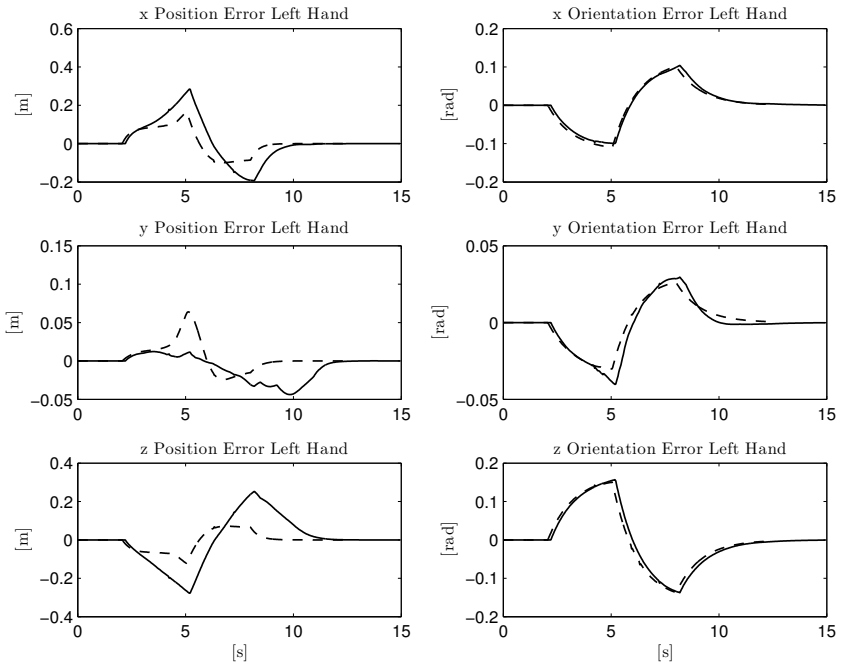


Fig. 3.14: Cartesian error on the left hand while performing the task without (dashed lines) and with (continuous lines) the robot dynamics constraint

Cartesian errors are shown in Fig. 3.14. Despite the Cartesian errors are small when not using the robot dynamics constraint, the robot falls while trying to keep the Cartesian error small due to high joint torques, that cause a big deflection on the SEA joints. With the constraint activated, the Cartesian errors are larger but the robot does not fall and the torques on the joints are inside the bounds (and so is the joint deflection).

Resources Allocation: Velocity Allocation When a high priority task saturates a given constraint, and in particular joint resources (i.e., joint velocity limits or joint torque limits) lower priority tasks will not have control authority to keep task error low. This situation is similar to what in real-time computing systems is called *starvation*. In these cases, a small compromise on high priority task performance will provide a big increment in the ability of the lower priority tasks to perform, even when the robot has a high degree of redundancy. Such a compromise is non-linear in nature, since it is due to saturation, and will imply that hard priorities between tasks will be enforced, *unless* the high priority task requires too many resources from the system. A possible solution in order to implement this scheme, is *resource allocation*. The constraints for each task will be *scaled*, where constraints for higher priority tasks will be more stringent to constraints of lower priority tasks. Such scaling σ_i, with

$$0 < \sigma_i \leq 1, \sigma_i \leq \sigma_{i+1} \tag{3.51}$$

where i is the task priority (from 1 to n), will be applied to the resource constraint on each task, and the maximum amount of resources *exclusively* allocated to a certain task will be

$$\rho = \sigma_i - \sigma_{i-1} \tag{3.52}$$

where $\sigma_0 = 0$. A demonstration of such scheme is illustrated in Figure 3.15 for the following IK problem

$$\begin{pmatrix} \left(T_{\substack{\text{Right}\\\text{Foot}}} + T_{\substack{\text{Left}\\\text{Foot}}}\right) \setminus \\ \\ \left(T_{\substack{\text{Right}\\\text{Wrist}}} + T_{\substack{\text{Left}\\\text{Wrist}}}\right) \end{pmatrix} << B_{\substack{\text{Joint Velocity}\\\text{Limits}}} \tag{3.53}$$

The Velocity Allocation (VA) scheme in implemented in **OpenSoT** via the `VelocityAllocation` utility, which gets as input the minimum joint velocity constraint and maximum velocity constraint for a certain stack, and will apply the minimum value to the highest priority task, the maximum value to the higher priority task, and will apply a value equal to

$$\text{res}_{\text{min}} + i \times \frac{\text{res}_{\text{max}} - \text{res}_{\text{min}}}{n - 1} \tag{3.54}$$

for all the others, so that

$$\rho_i = \frac{\text{res}_{\text{max}} - \text{res}_{\text{min}}}{\text{res}_{\text{max}} \times n - 1} \tag{3.55}$$

. The VA can then be modified manually on each individual task, as long as the scaling relation 3.52 is satisfied.

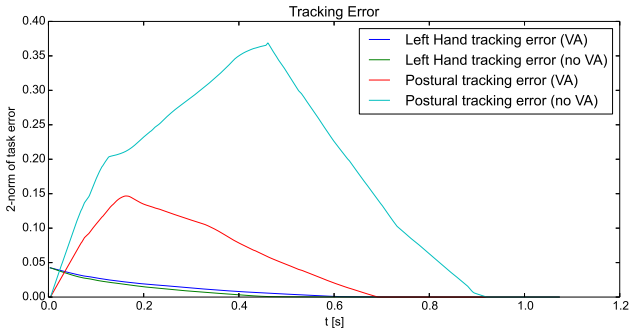


Fig. 3.15: When VA is active, all tasks in the stack have 0.15, 0.225, 0.3 rad/s limits, otherwise 0.3 rad/s for each task

Self Collision Avoidance During tele-operation tasks, a mistake in providing task references, or the combination of safe task trajectories with null-space tasks, can cause self-collisions. In particular, while the former case can be filtered by means of a check on the provided references, the latter is inherent in the idea of trajectory-less tasks (such as minimum effort, manipulation maximization, joint acceleration minimization), so that providing these tasks with self-collision awareness is necessary.

In Figure 3.16 a tele-operation task is executed with a self collision avoidance (SCA) constraint active on all the levels of the stack. The IK problem for the task is a whole-body task composed of

$$\begin{pmatrix} T_{\substack{\text{Right}\\\text{Foot}}} \\ T_{\text{CoM}_{XY}} \\ \left(T_{\substack{\text{Right}\\\text{Wrist}}} + T_{\substack{\text{Left}\\\text{Wrist}}}\right) \\ T_{\substack{\text{Joint}\\\text{Posture}}} \end{pmatrix} << \left(B_{\substack{\text{Joint}\\\text{Limits}}} + B_{\substack{\text{Joint Velocity}\\\text{Limits}}} + C_{SCA}\right) \tag{3.56}$$

The constraint activates on a certain pair of links when the distance between these two links is lower than a certain threshold. The maximum relative velocity between the two links in the direction of the minimum-distance segment between the two is then limited so to avoid interpenetration.

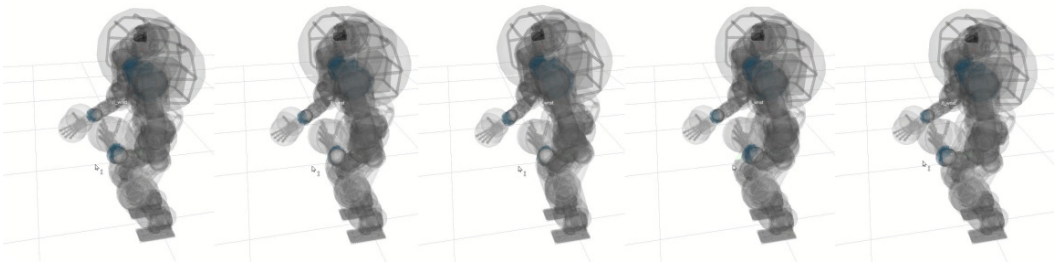


Fig. 3.16: Self Collision Avoidance (SCA) constraint on the Walk-Man robot. When the left wrist reference is too close to the body so as to cause a possible collision, the robot reconfigures itself by moving the waist back

Solver

In this section we show some results regarding the IK solver distributed with the **OpenSoT** library. In the first one we compare the results between the sparse and the dense implementation of the IK solver. In the second one we compare the quality of the generated joint trajectory changing the λ parameter.

Sparse implementation

Most of the time, the cost function 3.45 will result in a sparse matrix (in some particular cases this is not true, for instance when controlling the Center of Mass of the robot) with a dimension $m \times n$ where m is the dimension of the task and n is the number of the joints. It is possible to use the sparsity of the matrices in order to speed up the computation: in particular, *qpOASES* permits to define QP Problems in which

Hessian matrices are sparse. However we have tested the performances of the sparse implementation against the dense implementation (in the dense implementation we do not use special data structure for sparse matrices) and we found that, for a medium size IK Problem (29 variables, up to 63 constraints), the real advantage is in the *initialization* phase where the sparse solver is around twice faster than the dense solver. The IK Problem consists in 3 stacks:

$$\begin{pmatrix} T_{\substack{\text{Right}\\\text{Foot}}} \backslash \\ \left(T_{\substack{\text{Right}\\\text{Wrist}}} + T_{\substack{\text{Left}\\\text{Wrist}}} + T_{\text{Waist}} + T_{\text{Torso}}\right) \backslash \\ \left(T_{\substack{\text{Joint}\\\text{Posture}}} + T_{\substack{\text{Minimum}\\\text{Acceleration}}}\right) \end{pmatrix} << \left(B_{\substack{\text{Joint}\\\text{Limits}}} + B_{\substack{\text{Joint Velocity}\\\text{Limits}}} + C_{\substack{\text{CoM}\\\text{ConvexHull}}} + C_{\substack{\text{CoM Velocity}\\\text{Limits}}}\right) \tag{3.57}$$

Results (in seconds) are reported in Table 3.3. We see how the run takes around 0.5

Tab. 3.3: Sparse Vs Dense

	Initialization	Solve
Sparse	0.00143099	0.000490187
Dense	0.00297189	0.000475262

ms for both the solvers while the initialization takes around 1.5 ms for the Sparse solver and around 3 ms for the Dense solver. Figure 3.17 shows the trends of the *Sparse* solver Vs the *Dense* solver, the first one has a variance of 9.4777e-10, the latter of 2.3904e-09 (the *Sparse* solver is more constant). The solver can be clearly

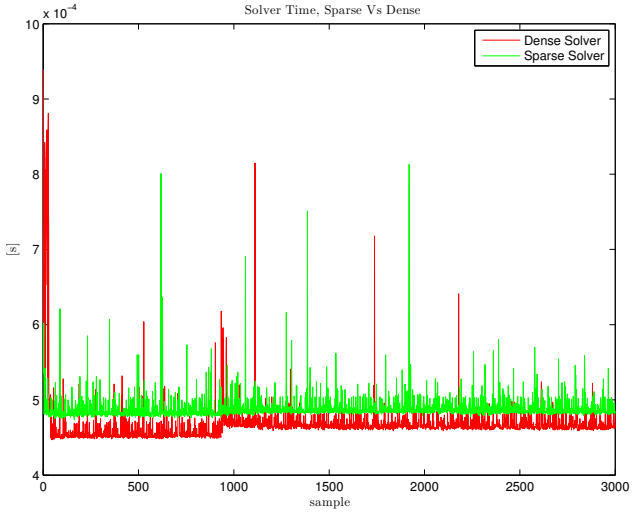


Fig. 3.17: Time needed to solve a stack of tasks, Sparse vs. Dense implementation

used for 1 ms real-time control.

Task execution and Regularisation

Here we show the effect of changing the regularisation term as well as the tracking performances in our IK solver. We consider a complex, whole-body, manipulation task composed by:

$$\begin{pmatrix} \left(T_{\substack{\text{Right}\\\text{Foot}}} + T_{\substack{\text{Joint}\\\text{Posture Torso}}}\right) \setminus \\ \left(T_{\substack{\text{Right}\\\text{Wrist}}} + T_{\substack{\text{Left}\\\text{Wrist}}} + T_{\text{Gaze}} + T_{\text{Waist}} + T_{\text{CoM}}\right) \setminus \\ \left(T_{\substack{\text{Joint}\\\text{Posture}}} + T_{\substack{\text{Min}\\\text{Acceleration}}}\right) \end{pmatrix} << \left(B_{\substack{\text{Joint}\\\text{Limits}}} + B_{\substack{\text{Joint Velocity}\\\text{Limits}}} + C_{\substack{\text{CoM}\\\text{ConvexHull}}}\right) \tag{3.58}$$

The *Gaze* task is built on top of the Cartesian task using the method presented in [Mil+11]. Some references, in terms of Cartesian trajectories (linear and circular) and constant poses for the end-effectors, are sent by the user. Computed joint trajectories are sent to the simulated robot integrating the initial value at each iteration (open-loop):

$$\mathbf{q}_{\text{k}} = \mathbf{q}_{\text{k-1}} + \boldsymbol{\Delta}\mathbf{q} \tag{3.59}$$

where $\boldsymbol{\Delta}\mathbf{q}$ is computed by our IK solver. Two experiments were performed changing the value of the λ in the regularisation term: the first one with $\lambda = 2.221 \cdot 10^{-3}$, the second with $\lambda = 2.221 \cdot 10^{-13}$. The results in Figures 3.18 and 3.19 clearly shows how this affects the resulting smoothness of the joints trajectories, as well as joint velocities and Cartesian error. In particular we see that high values of λ are associated with more smoothness of joint trajectories and velocities against higher values of Cartesian error during task execution. It is also clear how with high values of λ, joints velocities tend to saturate less. The average time that a solve takes, for the first run, is $8.4062 \cdot 10^{-4}$ $[s]$, while for the second is $8.0448 \cdot 10^{-4}$ $[s]$, the controller runs at 100Hz. The IK Problem has a total of 31 variables and up to 61 constraints.

3.1.6 The Darpa Robotics Challenge

In the DRC Finals the presented library was used to implement all the manipulation tasks (*driving*, *door opening*, *wall cutting* and *valve turning*) while keeping balance as well as considering joint limits and joint velocity limits. Two of them were performed during the finals: *driving* and *door opening*. For the *drive* task the IK problem used was:

$$\begin{pmatrix} T_{\substack{\text{Left}\\\text{Foot}}} \setminus \\ \\ T_{\substack{\text{Left}\\\text{Wrist}}} \end{pmatrix} << \left(B_{\substack{\text{Joint}\\\text{Limits}}} + B_{\substack{\text{Joint Velocity}\\\text{Limits}}}\right) \tag{3.60}$$

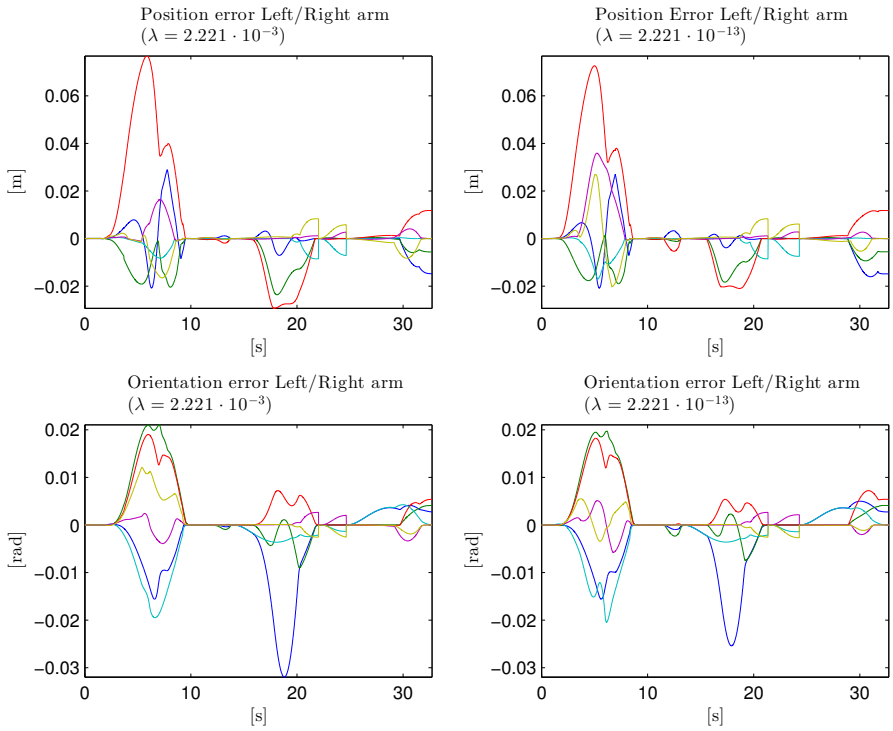


Fig. 3.18: Cartesian error for two different values of λ in task execution

For the *door* task the IK problem used was:

$$\begin{pmatrix} T_{\substack{\text{Right}\\\text{Foot}}} << C_{\substack{\text{CoM}\\\text{Velocity}}} \setminus \\ (T_{\text{Waist}} + T_{\text{CoM}}) << C_{\substack{\text{CoM}\\\text{Velocity}}} \setminus \\ \left(T_{\substack{\text{Left}\\\text{Wrist}}} + T_{\substack{\text{Right}\\\text{Wrist}}}\right) \setminus \\ T_{\substack{\text{Joint}\\\text{Postural}}} \end{pmatrix} << \left(B_{\substack{\text{Joint}\\\text{Limits}}} + B_{\substack{\text{Joint Velocity}\\\text{Limits}}}\right) \tag{3.61}$$

and velocity allocation strategy.

The IK solver was running in the on-board computer of the WALK-MAN robot and the trajectories were sent to the joint servos using (3.60). Joints were controlled using position control. WALK-MAN always accomplished the *driving* and *door opening* tasks (Figure 3.20) in both the runs of the DRC Finals[1].

Conclusions

This work introduced a novel whole-body control library called **OpenSoT**. Such framework allows to control a robot by solving a hierarchy of tasks with linear

[1] http://www.theroboticschallenge.org/finalist/walk-man

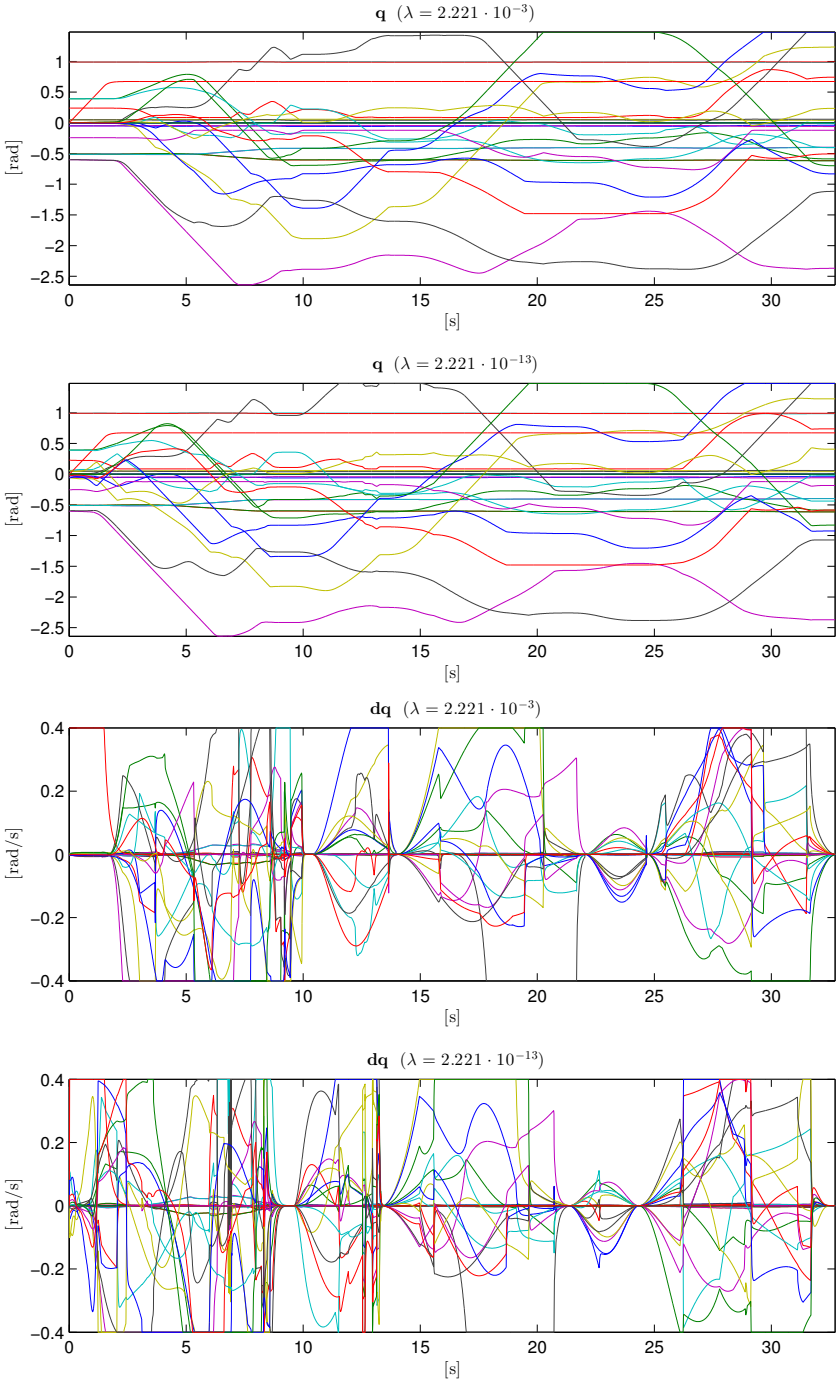


Fig. 3.19: Computed joint trajectories and joint velocities for two different values of λ in task execution. Notice how bigger λ values correspond to smoother trajectories (e.g. at secs $10 - 12$ and $23 - 24$)

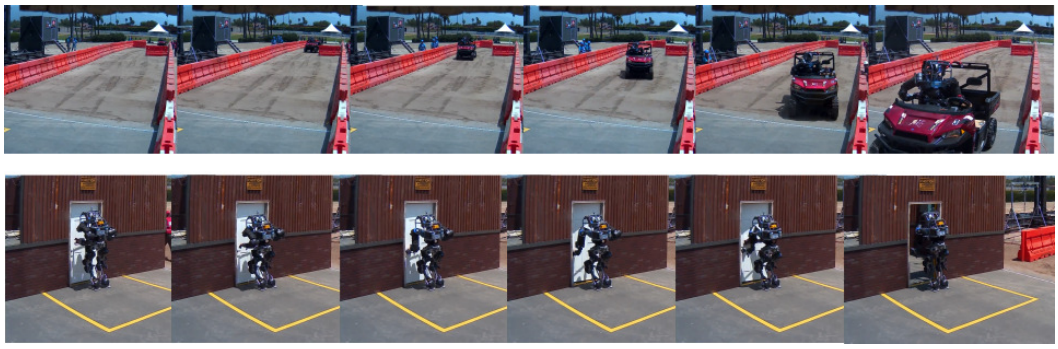


Fig. 3.20: WALK-MAN during DRC Finals completed two tasks: *driving* and *door opening*

constraints. Theoretical details about tasks, constraints and the IK solver available in the framework has been shown. In particular, the proposed IK solver satisfy the requirements in terms of robustness: singularity robustness, constraints/bounds specification and it permits to set priorities between tasks. The IK solver is composed by a *front-end*, that prepares the IK Problem for the QP solver which constitutes the *back-end*. As QP solver, a state-of-art library called qpOASES has been used. The set of tasks and constraints considered permits to perform high-level tasks without damaging the robot. Experiments presented in this section demonstrated the usefulness and the capabilities of the library.

We also presented a new constraint for the QP based IK, for fixed/floating base robots, that takes in consideration the robot dynamics to filter dynamically unfeasible motions. We presented the theoretical formulation and we showed results in simulation using our humanoid robot WALK-MAN considering a whole body task involving also other constraints such as joint limits, joint velocity limits, the Center of Mass of the robot has to lie inside the support polygon and self collision avoidance. We show that the robot dynamics constraint can make the task dynamically feasible and it is able to keep the torques at the torso on the given boundary limits. We think this is fundamental when working on real hardware as well as joint limits and joint velocity limits. This constraint is fundamental when Cartesian trajectories references are *aggressive*, for instance when considering motion capture data.

Future work will entail adding more tasks and constraints at the velocity level, add acceleration and torque tasks and constraints and explore different solvers.

This work is the second and last part in a series of papers introducing a novel whole-body control library called **OpenSoT**. Such framework allows to control a robot by solving a hierarchy of tasks with linear constraints. The framework provides a library of already implemented tasks, constraints, bounds and solvers as well as an easy way to implement new ones by providing common interfaces.

We addressed the theoretical foundations of the framework, and presented a large set of experiments with the intent of demonstrating the robustness of the library, its successful use in real-case scenarios, and the richness of features.

We showed how the **OpenSoT** framework contains a set of tools allowing the use of the library in conjunction with different robotic frameworks and high level programming languages, as well as facilitate the behcmarking of the resulting computed trajectories.
The performances of the library have been evaluated with a set of experiments and tests in multiple robotic platforms. **OpenSoT** was used by the WALK-MAN Team during the DRC Finals to perform all the manipulation tasks showing the possibility to use the framework in a real scenario.
Even if **OpenSoT** is a young project, it showed its ease of use by a large groups of first-time users with varying backgrounds and education levels that programmed tasks for the DRC Finals in a very short amount of time. Its flexible API allowed researchers[Fan+15] to easily extend it and successfully implement research code on top of a robust infrastructure.
The **OpenSoT** library is documented on the website and is available for download together with *iDynUtils* and the *capsule generator* on github, either as separate packages [MHR15; Fer+15; Rocb] or as part of our super-build system which takes care of installing the whole stack together with the necessary software dependencies [MH+].

Humanoids Software Architecture 4

In this section details about the software architecture used by the COMAN and WALK-MAN robot is introduced. In particular, this details the engineering efforts undertaken in view of the DRC competition in the software and control side. These details include considerations about control algorithms on the simulated vs physical robot, the software API and tools of the **OpenSoT** framework and in general for middle to high level control, and observations about the global humanoid architecture with a particular attention to the practical details required to produce robust incremental improvements in medium-sized teams. We also show a large set of examples of high-level task implementations as well as tests to evaluate the tooling against regressions, and in particular to test **OpenSoT**'s tasks, constraints and stacks (the highest level layer of the architecture which is automatically tested). This set of tests and examples demonstrate the performances of the architecture and the easiness of writing complex sets of tasks and constraints in the OpenSoT framework.

4.1 Development Pipeline

Part of the engineering work on the hardware and software for the DRC competition entry of the WALK-MAN robot involved organizing the efforts of a big team working under high pressure and delivering results in a very short amount of time. During one year and a half, a whole robot has been built from scratch, together with a new simulator, low level joint control, a high level control framework, a software component model and UI for teleoperation.

This work has been coordinated across IIT and two more universities in the consortium (Univerity of Pisa - UNIPI and Universitè Catholique de Louvain - UCL). Part of the work needed to feature such an achievement has involved adopting a shared development workflow, and in particular, an automated build system.

The *YCM* framework developed by the iCub facility department has been used for the WALK-MAN project, which allows to automatically manage the build order, the automatic downloading, compilation and installation of dependencies and the management of versions for each separate module in the build, and integrates seamlessly with the ROS build system, either *rosbuild* or *catkin*. The system allows to automatically install all the dependencies (binary or from source), to download, compile and install all the external libraries and the DRC modules in less than an hour, with a list of around 60 modules managed automatically. It allows to specify a profile (i.e. *ROBOT* and *SIMULATION* depending on the target location for the installation, be either a control computer or a development computer, where algorithms need to be tested in simulation - the different choice automatically sets up the environment to synchronize the algorithm's clock either with a wall

clock or simulation clock) and interest focus (allowing to automatically disable compilation of unneeded packages based on the user focus, be either control, sensing, robot model development, architecture and networking). The build system is well documented and provides a set of scripts that have been found to be of use to the team (e.g. setting up environment variables to use a remote simulation server, setting up a networked control architecture with a remote ROS or YARP master servers, install daemons to automatically install services useful for simulation servers or control computers, and advanced version management utilities which allow to bulk branch, tag/freeze, and analyze the version and changes on all the modules managed by the build system). Lastly, it allows for bulk version management via branching of the build system itself: in fact, the configuration file specifying the desired branch or tag to use for all the modules managed by the build system (`cmake/ProjectsTags.cmake`) is versioned itself. This allows to easily generate *versions* for the whole system, i.e. freeze notable working versions, such as a system used for a demo, or a system tuned with a certain network architecture, or with a certain version of the robot, or eventually even to manage codebases tuned for a partial robot build (as mentioned earlier, e.g. in the case for parallel development of lower-body and upper-body of a humanoid robot). A stripped-down version of the build system used for the DRC by the WALK-MAN team is hosted online [**robotology-superbuild**], and is being used as a basis for other projects in the team. Its documentation is available [**robotology-superbuild-handbook**] and is based on the original handbook developed for the team during the DRC.
The second fundamental aspect related to streamlining development and increasing productivity involved adopting a development workflow based on a series of fundamental steps:

- a very large set of unit and integration tests, especially for the core libraries and the high level control libraries. Regression testing has been a fundamental tool to quickly develop a consistent set of algorithms that evolved during time
- a continuous build system
- a shared GIT workflow with *master*, *devel* and *feature* branches, plus branches for each sprint (a so-called Agile development practice), or so called *sprint-week* that got merged at the end of each sprint

The GIT distributed source code managament system has been used on a self-hosted *gitlab* environment, which seved also as a hub for managing collaboration, by providing features for issue-tracking, code reviewing and documentation sharing via its WIKI system.

4.2 Component Model

In order to facilitate module development, a basic compoment model has been developed, the *Generic Yarp Module* (*GYM*), based on the YARP framework. Quoting

its webpage [**GYM**], *the Generic YARP Module (GYM) is a component model to easily develop software modules for robotics leveraging the YARP ecosystem*. It makes use of the *paramHelp* [**paramHelp**] library, which was designed to simplify the management of parameters of YARP modules and allows networked logging, configuration and (on-the-fly) reconfiguration. The library has been extended to automatically generate GUIs for online parameter tuning, with special focus on case scenarios most frequent in gain tuning (automatic generation of on-off switches for boolean parameters, sliders for integers and doubles in order to tune gains and algorithm strategies). GYM also provides facilities for automatic module creation, were several templates (or skeleton) can be chosen, i.e. helper modules (for developing robotics tools), modules for robot control (for developing robotics control and sensing algorithm that need need to run periodically and with a fixed rate, and need to directly access a part or the entirety of the robot hardware) and generic modules (for developing robotics library that don't need to access directly the robot hardware but are still periodic, e.g. trajectory generators and state machines).

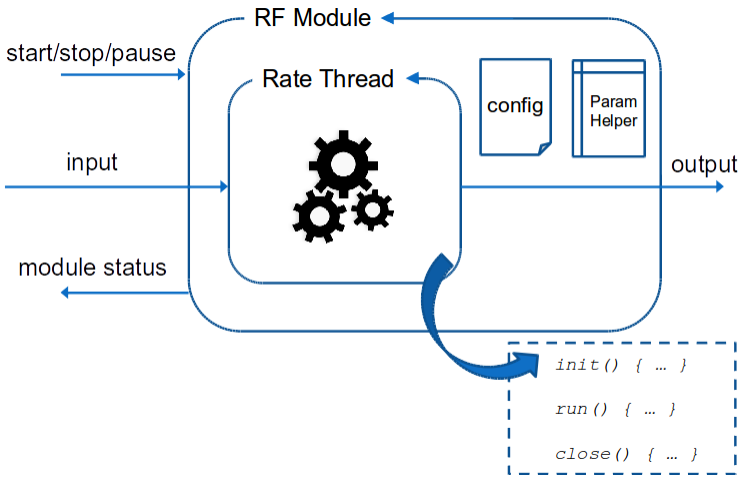
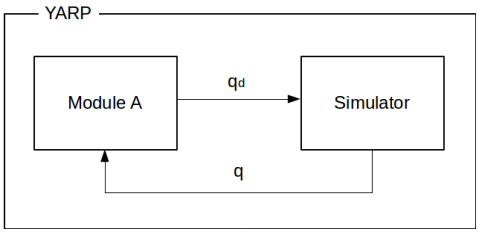


Fig. 4.1: structure of a GYM module

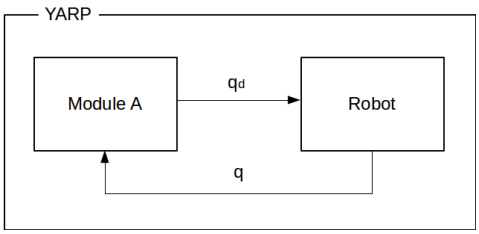
Every module is composed of at least two threads. A low frequency thread accepts high level management commands to `start`, `stop`, `pause`, `resume` execution of a module (the *switch* interface), gives information on the status of the module (the *status* interface), and provides a set of commands related to the *paramHelp* that allow to view, set and eventually store to disk all of the module's algorithms parameters. Additionally, each module has a main thread that is specific to the kind of module to implement and is usually a high frequency thread that takes care of the control and obtains sensor feedback from the robot and task level commands by the user.

4.3 Simulating and Controlling Compliant Robots

The simulator is a fundamental in the robot software development cycle as the first step to validate algorithms, thus minimizing the risks of accidents and hardware damage related to the application of dangerous whole-body behaviors. While low-



(a) Module A connected to the simulator



(b) Module A connected to the robot

Fig. 4.2: Module A writes desired joint position and read actual joint position without knowing if it is interfaced with the simulator or the robot since they expose the same interface.

level control can be extensively debugged on a testbench, the same is not true for more complex controls where failures both at the algorithmic and hardware level could cause unexpected behaviors. The *simulator* is a module that represents the real robot at the interface level (Figure 4.2). Such simulator module accepts control input (desired joint torques, desired joint position, ...) and outputs sensory feedback (cameras, joint positions, ...) from the simulated world. It allows to have both scripted and interactive simulations (human in-the-loop). One of the most important aspect of a simulator interface is the compatibility with the robot interface, in that a simulator should allow to develop modules that directly will work in the real robot without any need to rewrite code. In fact, when the physical robot is used, a software module should be able to replace the simulator by providing the same hardware interfaces (in our case, the interface provided by the YARP *robotInterface*).

With these concepts in mind we decided to extend one of the most known robotics simulator, Gazebo ([KH04]), to be compatible with one of the most used robotics framework, YARP, developed at the Italian Institute of Technology. While YARP is supported by the iCub simulator (iCubSim, [Tik+08]), that is a simulator specificly tailored to a specific platform. Gazebo, which has been recently chosen as the simulator for the DARPA Virtual Robotic Challenge (VRC, [DAR13]), aims to be a very generic tool for simulating different kind robots with a diverse set of sensors and actuation mechanisms, and allows the use of different dynamic engines, it is easily expandable through plugins and it has a strong and active community. Gazebo is maintained by the Open Source Robotics Foundation ([OSR11]).

It is useful to understand Gazebo plugins and YARP device drivers before describing the structure of our plugins (from now on *gazebo_yarp_plugins*).

Gazebo plugins are C++ classes that extend the functionalities of Gazebo, while YARP device drivers are C++ classes used in YARP for abstracting the functionality of robot devices. Usually, each class of gazebo_yarp_plugins embeds a YARP device driver in a Gazebo plugin.

Gazebo Plugins A plugin is a piece of code compiled as a shared library and loaded into the simulator. A plugin has direct access to all the functionalities of Gazebo from the physics engine to the simulated world. Furthermore, plugins are self-contained routines that are easily shared and can be inserted and removed from a running system. The algorithms and routines contained in such plugins can attached to several hooks in the simulation system, but the basic structure of a plugin requires an algorithm to be called in the moment when the simulation gets updated, usually corresponding to a simulation/integration step. There are 4 types of plugins in Gazebo: **world**, **model** and **sensor** plugins are attached to and control a specific simulated world(environment)/model(robot)/sensor respectively, while a **system** plugin is specified on the command line and is loaded during Gazebo startup.

YARP *Device Drivers* YARP provides special devices that act as proxies and make interfaces available through a carrier. The carrier can be implemented in several ways and in our architecture is mainly a network connection, via a tcp/ip or udp connection. This allows to seamlessly access devices remotely across the network. A device driver is a class that implements one or more interfaces. There are three separate points that cover the design and development of a devices in YARP:

- Implementing specific drivers for particular devices
- Defining interfaces for device families
- Implementing network wrappers for interfaces

For example the Control Board device driver implements a set of interfaces that are used to control the robot (IPositionControl, ITorqueControl, etc.) and another set of interfaces to read data from the motors (IEncoders, etc), while several interfaces can be wrapped to be accessed more conveniently through a single wrapped interface.

4.3.1 Gazebo-YARP Plugins

The suite gazebo_yarp_plugins (GYP) are composed of:

- Gazebo plugins that instantiate YARP device drivers,
- YARP device drivers that wrap Gazebo functionalities inside the YARP device interfaces.

When implementing a new robot model to be simulated via GYP, the robot topology needs to be implemented following the same design as that of the physical robot, and sensors and kinematic chains need to be named consistently with the physical robot, in order to obtain the same number of device drivers and the identical interface both for the simulated and physical robot. Other than the plugins, the gains for the

low level controller on the model need to be tuned to match those of the physical robot, with the due changes (i.e. low level PIDs on the simulated torque assume an ideal low-level torque control loop, and all low level control scheme based on PIDs command torques instead of voltages or currents). Parameters identification in simulation is still an open problem, with many quantitative questions regarding fidelity still open. Work is in progress to give solutions to the problem of simulation fidelity with notable recent results [**hauser13blem**; **uchida15**].

4.3.2 Robot and Simulation Description Formats: URDF, SRDF, SDF

Gazebo uses an XML-style format, Simulation Description Format (SDF), to save and load information about a simulated world or model. An SDF encapsulates all the necessary information for a simulation including the robot models, physics engine parameters, and plugins to be loaded (e.g. sensors plugins). The SDF for a simulation can be composed by loading external files, and the SDF for a robot can be compiled from its Universal Robot Description Format (URDF). Together with the Semantic Robot Description Format (SRDF), the URDF is gaining ground as a modern robot description format, which defines a (evolving) standard for robot description, which is not based on a minimal set of parameters, but allows to specify both the kinematic and dynamic properties of a model, together with a set of additional parameters such as sensors placement on the kinematic structure. The URDF of a robot can be extended with tags that are to be used by a simulator (e.g. friction parameters), or by third party tools. On the other hand, the SRDF provides semantic information about the robot, by defining the concept of kinematic chains and kinematic groups (e.g. used for planning or control using only a subset of joints), and allows to set up an allowed collision matrix (ACM) for self-collision detection. Like the URDF, the SRDF can be extended or used flexibly for a series of applications, e.g. it is possible to define sensor groups by providing a group with a list of links or joints to which sensors are attached. All the tools that have been developed for the DRC competition are based on the parsing of the URDF and SRDF file formats: the libraries computing kinematics and dynamics of the robot, as well as the interfaces connecting to the robot, all use the standard robot description models to load all the kinematics, dynamics, geometric, actuation and sensor information of the robot. Great care has been put into designing models which are modular and can be easily regenerated. Also, a set of tools which are going to be described later in this chapter, use the model to automatically compute capsules for collision detection, and ACM for self-collision avoidance.

4.4 Whole-Body Control

Despite the existence of many control libraries for robotics, the DRC experience highlighted how there is no clear winner in the field, either because of a lack of maturity of the software for real usage scenarios, or for a lack of flexibility. At the DRC finals only one team, *Team AIST-NEDO* [**Cisneros:15**] used the popular *Stack of Tasks* [**A-KanLamWie11**; Man+09], even though from the remaining 7 teams that published specifics about their architecture, other 3 [**hopkins:15**; **Feng2015-ru**; **ihmc_whole_body**] used a form of IK leveraging a QP formulation. Furthermore, team AIST-NEDO reported a failure on their software stack likely due to a malfunction with their Stack of Tasks, that caused sending a Not-A-Number (NAN) value to their controller and finally causing a failure, with consequent fall and hardware damage during the door task. To the knowledge of the author, no team used an architecture based on the famous iTasc approach [DS+07] or any of its extensions. We can also note that, regarding whole-body control, none of the popular control frameworks is interfaced with the popular *MoveIT* tool from the ROS ecosystem, underlining a divide in tooling between users and experts. This chapter complements the description of the framework presented in chapter [**sec:opensot-library**], and presents the software API and tools of the **OpenSoT** library. One of the major difference of **OpenSoT** framework with respect to other existing frameworks is the idea to define a meaningful API for experts, but also to *collect* a comprehensive library of implemented tasks, constraints and solvers that the end-user can employ. While experts can make use of the rich API to write their own tasks, constraints and solvers permits and easily enrich the existing set, end-users can either write their own stack or use one from the set of examples provided, in order to implement a high-level whole-body behavior. Another aim of concentrating on a standard API is to allow developers to easily benchmark the performance of different IK problems formulations and different solvers for the same high-level tasks.

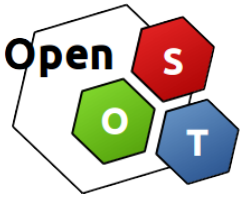


Fig. 4.3: OpenSoT Logo

A particular attention has been put also on the *toolset* that has been developed and made available together with the library: visual markers for the Cartesian control, bindings for different programming languages, auto-generation of capsules model for the Self Collision Avoidance (SCA) constraint, a Math of Task description language and an utility library, the *iDynUtils* library which takes care of wrapping several common robotics libraries to provide an easy-to-use comprehensive framework for robotics control (convex hull computation, self-collision checking, *SE3* math) and its complementary library, *RobotUtils* which allows to easily read sensors and send command to actuators in a minimum amount of code.

4.4.1 OpenSoT Software Architecture

Designing a good Software Architecture has been a crucial step in the development of **OpenSoT**. Firstly, the design has been focused on requirements of agile programming as commanded by the tight schedules of the development for the DRC competition. The software should be straightforward to install and to use, be robust both algorithmically and at the implementation level, and allow for ease of extendability, reusability and integration with existing frameworks.

Application Programming Interface (API)

The **OpenSoT** API is structured to favor reusability. Tasks are implemented through the class *Task* which provides an interface to obtain $\mathbf{A}$ and $\mathbf{b}$, a weight matrix W and a scalar weight λ for the task (`getA()`, `getb()`, `getWeight()`, `getlambda()`). Constraints and bounds are implemented in **OpenSoT** through the class *Constraint* which implements a simple interface providing equality constraints (`getAeq()`, `getbeq()`), inequality constraints (`getAeineq()`, `getbLowerBound()`, `getbUpperBound()`) and bounds (`getLowerBound()`, `getUpperBound()`).

Both *Task* and *Constraint* classes have a pure virtual `update()` method to be implemented. Most of the kinematics and dynamics information are directly available inside the *Task/Constraint* thanks to a pointer to the model of the robot. The robot model (see iDynUtils section) has to be updated at every control loop with measured joint position, velocities and force/torque, before the update of the IK problem as shown in Figure 4.4.

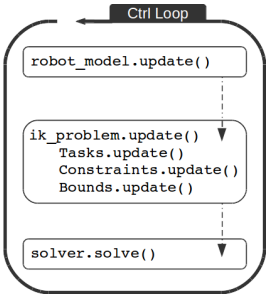


Fig. 4.4: At each control loop the robot model and the IK problem are updated and then the solver is called

Controlling a new robot with **OpenSoT** requires only writing a new stack (in most cases, a pre-made stack can be used for common whole-body tasks). Thanks to iDynUtils, every robot which is provided with an *URDF* and *SRDF* can be imported for use with the library.

OpenSoT provide also a simple base class to write a *Solver*. The main pure virtual methods consists in three constructors, depending if the solver permits to handle also bounds and constraints, and the `solve()` method.

Integration with Robotic Frameworks

By writing proper task wrappers it's possible to encapsulate the API and export the functionalities of **OpenSoT** over the network. Some wrappers are already available for the *ROS* and *YARP* middlewares. In particular, for *ROS* the Cartesian and joint postural tasks are wrapped so that it is possible to perform simple tele-operation through *rviz*'s *interactive markers*, as shown in Figure 4.5. Through *YARP* it's possible

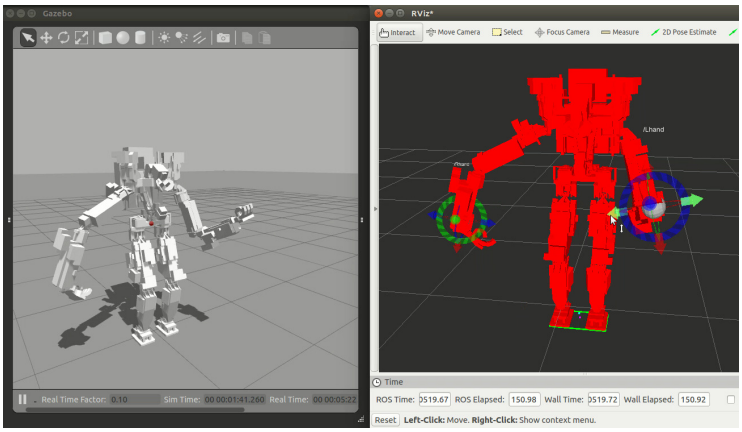


Fig. 4.5: The HYDRA robot tele-operated in simulation.

to change the parameters of all the tasks, including β and $\mathbf{W}$, and to get and set the task reference when contemplated (e.g., Cartesian, CoM, postural).

iDynUtils

As a support library to the **OpenSoT** framework we developed a whole-body control library named *iDynUtils*. This library is responsible for loading the model of the robot via the *iDynTree*[Nor+15a; Nor+15b] library.
The library is an interface and wrapper to other robotics libraries, and provides the following functionalities:

- robot pose estimate w.r.t. the world via forward-kinematics based dead-reckoning
- loop closures/kinematic constraints via kinematic tree rebasing, e.g. foot switching
- convex hull computation via *PCL*, management of link is contact with the environment with frictional constraints
- collision and self-collision checking via *Moveit!* and *fcl*, helper utilities to define collision white-lists, simplified bound in volumes support (capsules management)
- Cartesian utilities for *SE3* math, in particular quaternion implementation of 3.17

- interfaces for automatic transformation of frame of reference/poles for force/torque readings, and for frame of reference of IMU readings
- utility functions for management of the kinematic tree as a tree or as as a list of chains
- the utility class *RobotUtils* which allows to easily connect to a robot (physical or simulated), switch control mode, send actuator commands and read sensor data, and perform whole-body operations while retaining the possibility to address single kinematic chains indipendently

In particular with respect to item 1, 2 and 3, the structure of the library makes so that the dead-reckoning is a simple integrator based on the forward kinematics of the robot. In particular, among the links in contact with the environment, we select a link which will be called the *anchor* for the dead-reckoning phase, and a link to which the *floating base* will be attached. While the computation of the IK problem's solution can take into account explicitly the virtual joints of the floating base, most of the tasks already implemented in **OpenSoT** will follow an approach where the Jacobians are cut to take into account only the actuated joints. In this way, the solution of the problem will be computationally less intensive as the Hessian of the equivalent QP problem will be smaller, and we will drop the chain-closure constraint (i.e., imposing that the chain going from the *anchor* link to the world through the floating base virtual chain will have only internal movements, or alternatively, the twist of the *anchor* link with respect to the world frame will be 0). For this reason, the *iDynUtils* library has been implemented to provide functionalities for both use cases, and in particular the *anchor* and *floating base* links can be coincident, and during a typical walking phase, they will switch from one support foot to the other (with the switch command given by the walking algorithm), and remain constant during the double support phase.

Python Bindings

A set of bindings is provided for the **OpenSoT** framework that allow to send commands to an existing stack that runs as a server. While the stack needs to be defined and compiled in C++ at the moment, the approach allows to program the tasks in python using simple interfaces. In particular, python interfaces are available to allow the creator of python clients for the *YARP* bindings (introduced in 4.4.1) to **OpenSoT**. The python bindings ease the process of creating messages for the *YARP* interfaces, namely `pose` messages, `twist` messages, `joint position` messages and `trajectory` messages. Using these, the `YTask` python binding defines a generic binding to a task with a *YARP* interface (a *YTask*), such as getting and setting weights and gains. The `CartesianTask`, `CoMTask` and `PosturalTask` extend the base class `YTask` to provide specific functions to set and get references for the corresponding

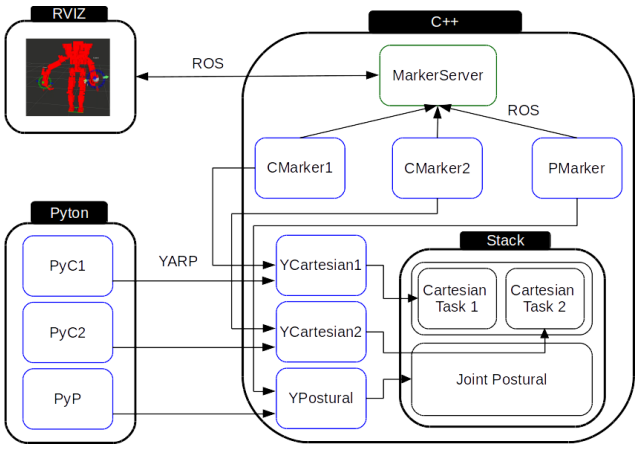


Fig. 4.6: Integration of Python and RVIZ markers inside **OpenSoT**

YARP interfaces to the corresponding tasks, as written in C++ (see Figure 4.6 for details). An example is provided to generate generic bindings via *SWIG* for interfacing with the *Klampt* simulator.

Integration with Generic Simulators

While **OpenSoT** has been used extensively with Gazebo, using an abstraction infrastructure that allowed to seamlessly switch from the physical robot to the simulated robot [MH+14], the **OpenSoT** library is generic and can be used with any simulation and control infrastructure. A suggested workflow for using **OpenSoT** with simulators and controllers written in C++ and python is provided in the examples. While the library iDynUtils is model-based, and the model is populated using URDF and SRDF descriptions of a robot, the main steps for connecting the library to a generic software is to specify conversion functions to transform joint positions and references between the simulator and the **OpenSoT**-based controller. The suggested way consists in transforming the joint reading vectors specified by the simulator in a joint readings map that associates the joint names with joint values, in a similar way to what *ROS* enforces via joint state messages and *YARP* implicitly does in the *Gazebo YARP plugins*. The same is done in **OpenSoT** to transform the joint readings map into its joint readings vector (in general the two vectors might assume different joint indices). Once specified this lightweight translation layer that uses sensors names to exchange data, **OpenSoT** can be encapsulated in the custom simulation/controller architecture. An example of a *HUBO* robot being simulated in the *Klampt* simulator

is shown in Figure 3.11, where the stack for the three tasks is a whole body stack composed of

$$\left(\begin{array}{c} \left(T_{\substack{\text{Right} \\ \text{Foot}}} + T_{\substack{\text{Left} \\ \text{Foot}}} \right) \backslash \\ T_{\text{CoM}_{\text{XY}}} \backslash \\ T_{\text{Waist}_{\text{Z}}} \backslash \\ \left(T_{\substack{\text{Right} \\ \text{Wrist}}} + T_{\substack{\text{Left} \\ \text{Wrist}}} \right) \backslash \\ T_{\substack{\text{Joint} \\ \text{Postural}}} \end{array} \right) << \left(B_{\substack{\text{Joint} \\ \text{Limits}}} + B_{\substack{\text{Joint Velocity} \\ \text{Limits}}} \right) \tag{4.1}$$

and velocity allocation strategy (see Resources Allocation: Velocity Allocation). The Hubo robot used in the simulation has a 1 DoFs torso, 6 DoFs legs and arms, 2 DoFs neck, 15 DoFs arms. Figures 3.11 are generated by using *Gazebo*. Notice how the two simulators use a different approach to controller simulation, whereas in *Klampt* we have a synchronous execution of the controller together with the simulator, while in *Gazebo* the controller runs asynchronously and needs to be synchronized by using the simulation clock, so as to obtain a simulated real-time execution on the controller.

Notes on Robustness

The **OpenSoT** framework has been developed with stability and robustness in mind, both at the algorithmic level, and at the software level. Best practices for software development are crucial when working in big groups of people under a tight deadline. For this reason, a set of unit and integration tests has been developed, which (where possible) also generate simulation data and graphs for human inspection. On the debugging side, the solver dumps the QP problem stack once a failure in solving a problem is encountered, which allows for inspection and reproduction.

Utilities

Other than the iDynUtils package, several utilities have been developed as **OpenSoT** ancillaries. Between these, the Capsule Generation toolkit and the Previewer class. The former allows to generate simplified bounding volumes for the robot links' geometries, as required by the collision avoidance task as defined in section Self-Collision Avoidance Constraint in 3.1.3. The latter allows to automatically check for a defined motion before executing it on the robot, by kinematically simulating a certain task while checking user-defined convergence and maximum error thresholds, and checking for self-collision. In particular, the Previewer is implemented to be used programatically to generate a Task workspace for a given stack and robot.

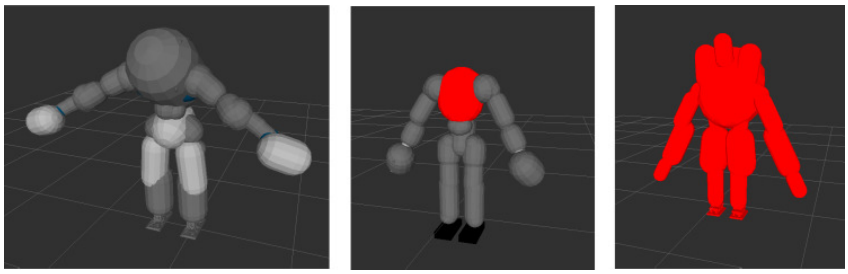


Fig. 4.7: Generated capsule models for WALK-MAN, COMAN and HYDRA respectively

Capsule Generation In order to compute the minimum distance between links, an automated way to generate capsule approximation of meshes for URDF model is needed. To this extent, we developed tools using the *Roboptim* [Mou+13; Mou+] numerical optimization for robotics library. Through the plugin *roboptim-capsule* the library finds the minimum volume capsule that contains the given polyhedra. The optimization problem resolving in optimal capsules is reported here for convenience [EK+13]

$$\begin{aligned} \min_{e1,e2,r} \quad & \|e_2 - e_1\|_2 \pi r^2 + \tfrac{4}{3}\pi r^3 \\ s.t. \quad & r - d(p, e_1 e_2) \geq 0, \qquad \vee p \in P \end{aligned}$$

where p is a vertex of the polyhedron P and we want to find the capsule of minimum volume for which the distance $(p, e_1 e_2)$ between p and the segment $e_1 e_2$ is smaller than the capsule radius.

The integration of the plugin with our architecture needs to circumvent the lacking of the capsule primitive in the URDF format and on all the ecosystem of libraries using the URDF format, including the popular interface *rviz*. Once our wrapper analyzes the URDF file of the model described with meshes, it will convert them to the corresponding capsules via the *roboptim-capsule* optimization, and create an URDF file with cylinders in place of meshes. The wrapper takes care to transform the capsule parameters between the endpoints-radius representation and radius, length, center representation during the process, in order to integrate the different components of the system. The self collision avoidance constraint then will scan the URDF looking for cylinders and interpret them as the body of a capsule, to feed the *fcl* library which takes care of computing minimum distances between the robot links using the simplified bounding volume. An example of application of this method on different robotic platforms can be seen in Figure 4.7. One last step after the generation of the capsule lies in creating the collision whitelists, that is, the default pairs of links to check for collision detection in the robot. This process is performed by the *moveit setup assistant*[Col+14] library, which randomly samples the joint configuration space, applies forward kinematics (FK) for each sample and finds link pairs which are always in collision and link pairs which are never in collision. These

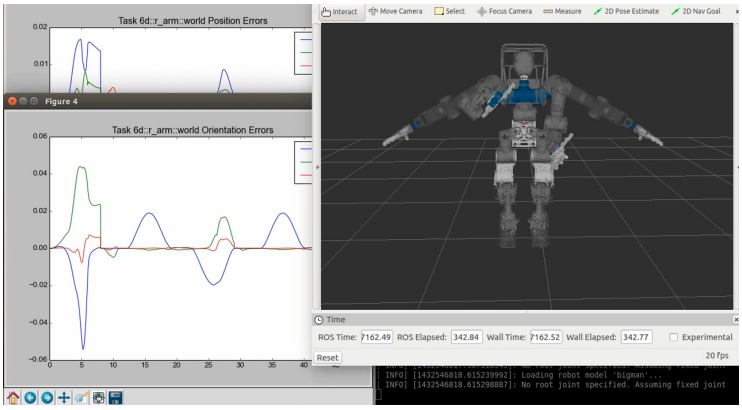


Fig. 4.8: The phantom model is superposed to the robot model to show the motion that the robot will perform. Plot of the trajectories are automatically generated by the previewer

lists are then stored into the SRDF file for later use by the self-collision avoidance algorithm: the link pairs which do not fall in this category are in fact put in the white-list for collision detection.

Previewer An architecture for previewing actions kinematically in Operational space before executing them on the robot (or in a dynamic simulation) is provided in order to automatically check for the feasibility of a certain movement. The interface of the *previewer* allows to *bind* a trajectory generator to a *Task*. During the execution of the task trajectory, the *previewer* will check several parameters to ensure that the task can be executed (Figure 4.8). In case the joint-space configuration diverges significantly from the previous one, it will perform a self-collision check. All these parameters can be tuned, and are exemplified in Table 4.1.
The *Previewer* will generate a results log containing error events, and can be configured with a callback, which can be used for example to publish the joint state of the robot while performing the action preview to *ROS*. The error log for the *Previewer* will contain events in the class

- collision
- Cartesian error unbounded

while the results log will keep track of the trajectory $(t, \mathbf{q})$ and of the Cartesian errors associated to the task execution along the trajectory, namely the moving average of the error norm, the current error norm, and the current Cartesian error, together with the Cartesian error to the goal (or final trajectory endpoint), where the Cartesian error is defined as in 3.17
The success of the action will be determined according to the settings of the previewer, and the specified duration of the preview action. In fact, a user can decide to preview an action partially, and get results from that partial preview of an entire trajectory, or preview an action till convergence is detected. The success is determined when no

error conditions were raised during preview, and one of the following convergence conditions is true at the end of the preview action:

- Cartesian error is small
- Cartesian error is small and not decreasing
- Cartesian error is small, not decreasing, and joint velocities are below a user-defined threshold (null-space tasks converged)

Constant	Default Value	Meaning
`TRAJECTORIES_EXPIRED_SAFETY_MARGIN`	1.5	when previewing for an infinite time, stop `if t >= longest_task_duration * safety_margin`
`UPDATE_ERR_STATS_WINDOW_SIZE_IN_SECS`	0.1	set the time window of the moving average filter for the error statistics
`PREVIEWER_CHECK_MAX_RETRIES`	3	if `solve()` fails, retry for a maximum of `MAX_RETRIES`
`TRAJ_BINDING_MAXIMUM_ALLOWED_ERR`	5×10^{-2}	maximum allowed Cartesian error, in norm, during task execution
`TRAJ_BINDING_CONVERGENCE_TOLERANCE`	5×10^{-4}	threshold value for Cartesian error norm convergence

Tab. 4.1: Previewer default configuration. Values used for previewing motions on an adult-sized compliant humanoid robot.

Logging and Plotting YARP provides a functionality for plotting and logging signals, such as joint velocities, joint torques and force/torque sensor readings. The same can be said about ROS, and since the software architecture used by the WALK-MAN team comprised a mix between the two middlewares, both could be used to store and visualize crucial data during robot operation. Still, the facilities these middlewares provide expect developers to *publish* (via sockets or other carriers) all the data that needs to be stored/visualized, resulting in a large overhead which often can become unbearable for continuous logging by the onboard control pc, especially when, together with the basic sensor data, we are interested in logging and visualizing statistics about the control algorithm performance.
For this reason, a toolkit to easily store and plot data has been implemented in OpenSoT. The logging and plotting framework is based on the idea of *flushers* and *plotters*. The *logger* is created by stacking a list of flushers, each of them taking care of serializing the data to be saved on a text file. Each flusher can be instantiated with additional information regarding the data that it's storing, e.g. joint names and unit of measurements. A plotter can be created that plots the logged data, a subset of the data, or the result of an operation executed on the stored data (such as norm, multiplication between compatible data fields and filtering, computation of cartesian error), and in general a plot will consist of several plotters taking care of visualizing the data saved by a logger. The plotter can make use of the semantic

information about the data form the flushers, so that many aspects of plot generation can be automated, e.g. legend generation. The framework allows to automatically store and plot data relative to Tasks, Constraints, robot sensors data, and raw data (scalars, vectors, matrices); the user can easily write additional flushers and plotters for arbitrary complex data structures, extend the basic flushers in order to store data using different data formats, or extend the basic plotters in order to plot using different plotting environments. At the time the flushers and the plotters all use python data and plotting functionalities (via *pyplot*), allowing the plotters to work even on the control pc of the robot where *Matlab* is not available.

4.5 Conclusions

While in previous section the theoretical foundations of the high level control framework are addressed and presented together with a large set of experiments, in this section the API, the software infrastructure and the utilities ecosystem has been detailed, with the intent of demonstrating the richness of features of our high level control scheme. Statistics and best-practices used in the WALK-MAN team have been presented as part of the work for the DRC competition. Lastly, the simulation, middleware and component model used by the team has been presented as a fundamental scaffolding of the work of the team and as a means of coordinating its work.

Conclusions

5

In this work, part of the results of the effort on the Darpa Robotics Challenge for team WALK-MAN have been presented. Some of the results have been obtained in an attempt to engineer the state of the art in humanoid robotics. As previously highlighted, even though the research work in the humanoid robotics field is varied and is carried over by many laboratories around the world following different approaches, one of the surprising points of the competition lied in the fact that the final control architecture adopted by many teams tended to be very similar. One could argue how this showed a significant cap put by technology on the physical implementation of the state of the art of robotic control and planning.
On the more generic topic concerning the coordination of the team for the DRC competition, a significant experience has been gained on the field:

- we learnt that most humanoid robots are more robust than what we expected, with lots of falls during the Darpa Robotics Challenge not being fatal to the hardware. In particular, the WALK-MAN robot fell graciously, in part thanks to its actuation design, and did not suffer significant damage to the structure, actuators or sensors. The foam padding and the cage structure protecting the head also played an important role for the structural integrity of the robot after falling.

- leading change is very difficult. Streamlining a development cycle has encountered the resistance of the group initially, while has proven fundamental and much appreciated later on. In particular, switching from a hand-made per-module development workspace to an automatic building system with dependency resolution (using YCM) has been a great challenge.

- one of the critical points in our software infrastracture revealed itself at the wake of the DRC finals: for performance reasons, the initial design consisting of one YARP control board for every kinematic chain of the robot (which implies creating 6 threads on the control computer, with 6 different set of ports streaming sensors data and receiving input commands for all the joints of that particular kinematic chain) has been substituted with a final setup with just a single Control Board for every joint in the robot. As a result, while having multiple kinematic chains allowed to control the robot even with missing limbs (i.e., during the initial development phase, were using only the lower body or only the upper body gave chance to the manipulation and locomotion teams to work independently as if having two different robots), the final setup was less demanding of the limited resources of the on-board computer and allowed jitter-less 1Khz control on the low-latency Linux kernel. This implied also

adapting the *RobotUtils* middleware to the new network configuration, as the middleware was not updated on time to be topology-agnostic(that is, to be configurable for different Control Board designs) - finally the middleware had to be hardcoded to support the new robot configuration, with severe drawbacks in the overall software architecture and consequently on the productivity of researchers using the software stack (e.g., the need to change the code when switching from simulation to physical robot). Thus, the lesson learnt was that configurability of topology at every layer in the architecture is very important, especially when building a new robot, were computing requirements and kinematic and network topology might change due to maintenance, changing requirements or debugging.

Regarding simulation, we had very promising results and had a great adoption rate to the simulator and relative tooling, which soon become irreplaceable. Researchers experience great benefits from previewing their work with a visual, dynamic simulator, where the code used to move the robot in simulation was the same used for the physical robot. Ongoing work is progressing in the study of more faithful simulation techniques for compliant robots, were also the dynamics of the motor in a SEA joint are taken into account. Still, a significant amount of work should be invested in obtaining proper simulation tools for underactuated and compliant hands, in particular questions arise on the grade of fidelity one might obtain with such methods, and to answer this question we should first resort to fine identification of the hand model used in simulation, together with some fundamental object/hand properties like friction. This remains an open area of work.
On the OpenSoT front, the experience during the development and use of the library have been very valuable:

- even though a library is built with ambitious goals for flexibility, adaptability and sharability, in fact the research community is not apt to adopting standards which are built by possible competitors, unless it is addressed specifically in a formal scientific collaboration. On the other side, the ROS ecosystem over the years gained increasingly more ground as a platform for sharing software modules, and as a means to compose them to obtain high level, complex robotic systems. Of course, in general the community of researches expert in one field and the community of users of the products of that field belong to two different scenarios that intersect only in some circumstances. One idea for the future is to export the OpenSoT functionalities as a MoveIt plugin, in order to provide a robust whole-body framework which is ready to use for the large community of ROS users.

- it is very difficult in the research community to make sound statements about the robustness of software. In some cases, the software is seen as a byproduct of a theoretical development in a certain field, so that criticizing the software

in terms of robustness it's often not possible or easy to do. A large part of the community can recognize a strong theoretical contribution, but fails at recognizing the value of research work in applied engineering. The Darpa Robotics Challenge very clearly pushed the community to distinguish the research and technology that works reliably in the physical world from that which belongs only to the lab. More and more robotics challenges are being organized yearly, with the hope that the community will gain more awareness and sensibility to the problem of implementing research on a physical robot.

- while the field of resolved velocity control is still not exhausted (see recent results in learning, task specification, robust constraint specification [**del2015robustness**] in constrained control problems), the next step in frameworks like OpenSoT is not strictly in control, but in sensing. Many groups work in sensing and estimation, and in order to obtain more robust control it is important to have good state observers and estimators. Our experience showed that without filtering, tools like the dynamics constraint implemented in our robots can show jittety behaviors. At the moment, the solution in this cases where good joint velocity and torque estimates was needed, was to set gains which were more forgiving, or to relax the constraints.

- while many might argue that different formulations of the hiearchical constrained resolved velocity control might result in an unfeasible problem (e.g. hard constraints vs slack variable minimization), very often the problem of unfeasibility boils down to being a design decision rather than a formulation problem. For example, even when the constraints are hard, meaning that breaking those contraints results in obtaining no solution from the IK solver, the unfeasibility problem has a physical meaning which is not just mathematical, and entails practical questions such as: what strategy should to adopt when it's detected that the robot is going to fall, what should happen when (for whichever reason) the joint configurations are found to be out of the designed safety limits, or when joints are exerting torques which are greater than planned? In some of these cases, in our experience we kept unfeasibility by design, in some other cases we relaxed a constraint. More in practice, in OpenSoT the joint limit constraints are hard constraints, so in some circumstances, because of the compliant joints, the robot might start in a configuration from which no solution to the IK is feasible, and contingency actions can be taken after an human operator verifies the situation. On the other side, the dynamics constrained has been relaxed so as to be always feasible. In fact, the dynamics constraint is in general a *repulsive* constraint, meaning that even when the IK solution minimizing the task error might impose a joint to move in a certain direction, the constraint might ask the joint to move in the opposite direction. Sometimes, the speed at which the constraint imposes the joint movement is too high and goes in conflict with the joint velocity limits, making the problem

unfeasible. For example, imagine having a pendulum with a heavy weight attached to its extremity, and a torque limit constraint on its joint. When the pendulum is at rest at the configuration of maximum torque, the torque limit might ask us to accelerate downwards in order to decrese the load on the motor. Still, the requested movement might be unfeasible for the joint velocity limit constraint. In a way, this highlights the need of enforcing priority even between constraints. We implemented this by relaxing the dynamics (or torque limits) constraint as to alway impose velocities which are also feasible for the higher priority constriants (which, in this case, is the joint velocity limits).

- while the admittance control scheme that is implemented by the *Interaction* task (paragraph 3.1.3) works well in most cases, it still does not work for controlling ground reaction forces when in double support phase with no other external contacts. A more generic approach should be considered to obtain either whole-body hybrid position/force control, or force and position control through a whole-body admittance or compliance control.

- while task specification is relatively easy to implement, gain tuning and priority tuning is a major problem when designing a stack. Furthermore, task designers require in some cases stacks able perform a particular motion or a set of motions in a human-like manner, and while we can anticipate this need will become more and more significant as service robotics progresses into the house, it is difficult or impossible to design human-like motions in terms of simple cost functions and hand-tuning of gains. In the same way, it is relatively difficult to design stacks that perform well globally, or in other words, trajectories should not be generated exclusively by a local optimization method (like the one used in OpenSoT), but a planner should provide setpoints for the controller. In both cases, planning of the gains and priorities could be an area of work, be either by means of optimal control, sampling-based planning or machine learning techniques.

Future work for the framework might include implementing more powerful minimum effort and posture optimization tasks, based on grasp theory. Preliminary results show the possibility to implement a constrained minimum effort task that minimizes the effort of actuators in face of any expected or measured force in any point of the body of the robot.

Bibliography

[ADS14] E Aertbelien and J De Schutter. „eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs“. In: *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. 2014, pp. 1540–1546 (cit. on p. 33).

[Ajo+14] A Ajoudani, Jinoh Lee, A Rocchi, et al. „A manipulation framework for compliant humanoid COMAN: Application to a valve turning task“. In: *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*. Nov. 2014, pp. 664–670 (cit. on p. 29).

[BA] Patrick Beeson and Barrett Ames. „TRAC-IK: An Open-Source Library for Improved Solving of Generic Inverse Kinematics“. In: *Humanoids 2015*.

[BB07] Adrian Boeing and Thomas Bräunl. „Evaluation of real-time physics simulation systems“. In: *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*. ACM. 2007, pp. 281–288 (cit. on p. 16).

[Bir11] L Birglen. „From flapping wings to underactuated fingers and beyond: a broad look to self-adaptive mechanisms“. In: *Mechanical Sciences* 2.1 (2011), pp. 5–10 (cit. on p. 17).

[Bon+14] M Bonilla, E Farnioli, C Piazza, et al. „Grasping with Soft Hands“. In: *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*. IEEE. 2014, pp. 581–587 (cit. on p. 15).

[Cal+08] D. Calisi, A. Censi, L. Iocchi, and D. Nardi. „OpenRDK: a modular framework for robotic software development“. In: *Proceedings of International Conference on Intelligent Robots and Systems (IROS)*. Nice, France, Sept. 2008, pp. 1872–1877 (cit. on p. 5).

[Cat+14] Manuel G Catalano, Giorgio Grioli, Edoardo Farnioli, et al. „Adaptive synergies for the design and control of the Pisa/IIT SoftHand“. In: *The International Journal of Robotics Research* 33.5 (2014), pp. 768–782 (cit. on p. 14).

[Chi+91] Pasquale Chiacchio, Stefano Chiaverini, Lorenzo Sciavicco, and Bruno Siciliano. „Closed-loop inverse kinematics schemes for constrained redundant manipulators with task space augmentation and task priority strategy“. In: *The International Journal of Robotics Research* 10.4 (1991), pp. 410–425 (cit. on pp. 32, 34).

[Col+14] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. „Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study“. In: (Apr. 2014). arXiv: `1404.3785 [cs.RO]` (cit. on p. 73).

[Cor] MSC Software Corp. *Adams*. http://www.mscsoftware.com/product/adams. Accessed: 2015-08-26 (cit. on p. 15).

[DAR13] DARPA. *DARPA Robotics Challenge*. http://www.theroboticschallenge.org/. Accessed: 2016-1-20. 2013 (cit. on p. 64).

[DB13] R. Deimel and O. Brock. „A compliant hand based on a novel pneumatic actuator“. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. May 2013, pp. 2047–2053 (cit. on p. 14).

[DB14] Raphael Deimel and Oliver Brock. „A novel type of compliant, underactuated robotic hand for dexterous grasping“. In: *Robotics: Science and Systems, Berkeley, CA* (2014), pp. 1687–1692 (cit. on p. 14).

[Dec+09] W Decre, R Smits, H Bruyninckx, and J De Schutter. „Extending iTaSC to support inequality constraints and non-instantaneous task specification“. In: *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*. May 2009, pp. 964–971 (cit. on p. 32).

[Dec+13] W Decre, H Bruyninckx, and J De Schutter. „Extending the iTaSC Constraint-based Robot Task Specification Framework to Time-Independent Trajectories and User-Configurable Task Horizons“. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. May 2013, pp. 1941–1948 (cit. on p. 32).

[Dia10] Rosen Diankov. „Automated Construction of Robotic Manipulation Programs“. PhD thesis. Carnegie Mellon University, Robotics Institute, Aug. 2010 (cit. on p. 6).

[DK08] Rosen Diankov and James Kuffner. „OpenRAVE: A planning architecture for autonomous robotics“. In: *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34* 79 (2008) (cit. on pp. 3, 15).

[DP+14] Andrea Del Prete, Francesco Romano, Lorenzo Natale, et al. „Prioritized optimal control“. In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. 2014, pp. 2540–2545 (cit. on p. 32).

[DS+07] Joris De Schutter, Tinne De Laet, Johan Rutgeerts, et al. „Constraint-based Task Specification and Estimation for Sensor-Based Robot Systems in the Presence of Geometric Uncertainty“. In: *The International Journal of Robotics Research* 26.5 (2007), pp. 433–455 (cit. on pp. 32, 67).

[EK+13] Antonio El Khoury, Florent Lamiraux, and Michel Taix. „Optimal motion planning for humanoid robots“. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. May 2013, pp. 3136–3141 (cit. on pp. 43, 73).

[Elj+14] Jorhabib Eljaik, Andrea del Prete, SIlvio Traversaro, Marco Randazzo, and Francesco Nori. „WBI Toolbox (WBI-T): A Simulink Wrapper for Robot Whole Body Control.“ In: *ICRA, Workshop on MATLAB/Simulink for Robotics Education and Research*. IEEE, 2014 (cit. on p. 11).

[Ere+15] T. Erez, Y. Tassa, and E. Todorov. „Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX“. In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. May 2015, pp. 4397–4404 (cit. on p. 16).

[Erw03] Coumans Erwin. *Bullet Physics Engine*. http://www.bulletphysics.org/. Accessed: 2016-1-20. 2003 (cit. on pp. 6, 14).

[Esc+14] Adrien Escande, Nicolas Mansard, and Pierre-Brice Wieber. „Hierarchical quadratic programming: Fast online humanoid-robot motion generation“. In: *The International Journal of Robotics Research* 33.7 (June 2014), pp. 1006–1028 (cit. on pp. 32, 34).

[Fan+15] Cheng Fang, Alessio Rocchi, Enrico Mingo Hoffman, Nikos G. Tsagarakis, and Darwin G. Caldwell. „Efficient Self-Collision Avoidance based on Focus of Interest for Humanoid Robots“. In: *Humanoid Robots (Humanoids), 2015 15th IEEE-RAS International Conference on*. Korea, 2015 (cit. on pp. 29, 60).

[FC92] Carlo Ferrari and John Canny. „Planning optimal grasps“. In: *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*. IEEE. 1992, pp. 2290–2295 (cit. on p. 15).

[FDL14] Fabrizio Flacco and Alessandro De Luca. „A reverse priority approach to multi-task control of redundant robots“. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*. 2014, pp. 2421–2427 (cit. on p. 32).

[Fea07] Roy Featherstone. *Rigid Body Dynamics Algorithms*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007 (cit. on p. 6).

[Fer+13] Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. „qpOASES: A parametric active-set algorithm for quadratic programming“. In: *Mathematical Programming Computation* (2013), pp. 1–37 (cit. on pp. 31, 45).

[Fer+15] Mirko Ferrati, Enrico Mingo Hoffman, and Alessio Rocchi. *iDynUtils webpage*. `https://github.com/robotology-playground/idynutils`. Accessed: 2015-11-27. 2015 (cit. on p. 60).

[Fla+12] Fabrizio Flacco, Alessandro De Luca, and Oussama Khatib. „Motion control of redundant robots under joint constraints: Saturation in the Null Space“. In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. May 2012, pp. 285–292 (cit. on p. 35).

[Fok+15] Chien-Liang Fok, Gwendolyn Johnson, Luis Sentis, Aloysius Mok, and John D Yamokoski. „ControlIt! — A Software Framework for Whole-Body Operational Space Control“. In: *International Journal of Humanoid Robotics* 0.0 (Oct. 2015), p. 1550040 (cit. on p. 33).

[FT87] B. Faverjon and P. Tournassoud. „A local based approach for path planning of manipulators with a high number of degrees of freedom“. In: *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*. Vol. 4. Mar. 1987, pp. 1152–1159 (cit. on p. 43).

[Ger+03] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. „The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems“. In: *In Proceedings of the 11th International Conference on Advanced Robotics*. 2003, pp. 317–323.

[Gri+12] Giorgio Grioli, Manuel Catalano, Emanuele Silvestro, Simone Tono, and Antonio Bicchi. „Adaptive synergies: an approach to the design of under-actuated robotic hands“. In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE. 2012, pp. 1251–1256 (cit. on pp. 17, 18).

[Hau13] Kris Hauser. „Robust contact generation for robot simulation with unstructured meshes“. In: *International Symposium on Robotics Research, Singapore*. 2013 (cit. on pp. 14, 15, 18).

[Hau14] K Hauser. „Fast dynamic optimization of robot paths under actuator limits and frictional contact“. In: *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. ieeexplore.ieee.org, May 2014, pp. 2990–2996 (cit. on p. 32).

[HP14] John M Hsu and Steven C Peters. „Extending Open Dynamics Engine for the DARPA Virtual Robotics Challenge“. In: *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 37–48 (cit. on pp. 3, 6).

[Iva+14] Serena Ivaldi, Vincent Padois, and Francesco Nori. „Tools for dynamics simulation of robots: a survey based on user feedback“. In: *CoRR* abs/1402.7050 (2014) (cit. on p. 5).

[Kaj+03] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, et al. „Biped walking pattern generation by using preview control of zero-moment point.“ In: *ICRA*. IEEE, 2003, pp. 1620–1626.

[Kan+04] Fumio Kanehiro, Hirohisa Hirukawa, and Shuuji Kajita. „OpenHRP: Open Architecture Humanoid Robotics Platform.“ In: *I. J. Robotic Res.* 23.2 (2004), pp. 155–165 (cit. on p. 6).

[Kan+08] Fumio Kanehiro, Florent Lamiraux, Oussama Kanoun, Eiichi Yoshida, and Jean-Paul Laumond. „A local collision avoidance method for non-strictly convex polyhedra“. In: *Proceedings of robotics: science and systems IV* (2008) (cit. on p. 43).

[Kan+09] Oussama Kanoun, Florent Lamiraux, Pierre-Brice Wieber, et al. „Prioritizing linear equality and inequality systems: application to local motion planning for redundant robots“. In: *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE. 2009, pp. 2939–2944 (cit. on p. 32).

[Kan+11] Oussama Kanoun, Florent Lamiraux, and Pierre-Brice Wieber. „Kinematic Control of Redundant Manipulators: Generalizing the Task-Priority Framework to Inequality Task“. In: *Robotics, IEEE Transactions on* 27.4 (2011), pp. 785–792 (cit. on p. 34).

[Kap+15] Daniel Kappler, Jeannette Bohg, and Stefan Schaal. „Leveraging big data for grasp planning“. In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE. 2015, pp. 4304–4311 (cit. on pp. 14, 15).

[KH04] Nathan Koenig and Andrew Howard. „Design and use paradigms for gazebo, an open-source multi-robot simulator“. In: *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*. Vol. 3. 2004, pp. 2149–2154 (cit. on pp. 6, 64).

[Kim+13] Junggon Kim, K. Iwamoto, J.J. Kuffner, Y. Ota, and N.S. Pollard. „Physically Based Grasp Quality Evaluation Under Pose Uncertainty“. In: *Robotics, IEEE Transactions on* 29.6 (Dec. 2013), pp. 1424–1439 (cit. on pp. 14, 15).

[Kne14] K. Knese. „Realizing Online (Self-)Collision Avoidance Based on Inequality Constraints with Hierarchical Inverse Kinematics“. MA thesis. Germany: Technical University of Munich, July 2014 (cit. on p. 43).

[Lee+14] Jinoh Lee, A Ajoudani, E M Hoffman, et al. „Upper-body impedance control with variable stiffness for a door opening task“. In: *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*. ieeexplore.ieee.org, Nov. 2014, pp. 713–719 (cit. on p. 29).

[Leó+10] Beatriz León, Stefan Ulbrich, Rosen Diankov, et al. „OpenGrasp: a toolkit for robot grasping simulation“. In: *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2010, pp. 109–120 (cit. on pp. 3, 15).

[Liu+15] Mingxing Liu, Yang Tan, and Vincent Padois. „Generalized hierarchical control“. In: *Auton. Robots* (2015), pp. 1–15 (cit. on p. 32).

[Ma+13] R.R. Ma, L.U. Odhner, and A.M. Dollar. „A modular, open-source 3D printed underactuated hand“. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. May 2013, pp. 2737–2743 (cit. on p. 14).

[MA04] A.T. Miller and P.K. Allen. „Graspit! A versatile simulator for robotic grasping“. In: *Robotics Automation Magazine, IEEE* 11.4 (Dec. 2004), pp. 110–122 (cit. on pp. 3, 15).

[Man+09] N Mansard, O Stasse, P Evrard, and A Kheddar. „A Versatile Generalized Inverted Kinematics Implementation for Collaborative Working Humanoid Robots: The Stack of Tasks“. In: *International Conference on Advanced Robotics (ICAR)*. June 2009, p. 119 (cit. on pp. 31, 67).

[Men] Samir Menon. *Standard Control Library*. `http://web.stanford.edu/~smenon/scl.html`. Accessed: 2015-11-2 (cit. on p. 33).

[Met+06] G. Metta, P. Fitzpatrick, and L. Natale. „YARP: Yet Another Robot Platform“. In: *International Journal of Advanced Robotics Systems, special issue on Software Development and Integration in Robotics* 3.1 (2006) (cit. on p. 5).

[MH+] Enrico Mingo Hoffman, Luca Muratore, and Alessio Rocchi. *robotology-superbuild webpage*. `https://github.com/robotology-playground/opensot`. Accessed: 2015-11-27 (cit. on p. 60).

[MH+14] Enrico Mingo Hoffman, Silvio Traversaro, Alessio Rocchi, et al. „Yarp based plugins for gazebo simulator“. In: *Modelling and Simulation for Autonomous Systems: First International Workshop, MESAS 2014, Rome, Italy, May 5-6, 2014, Revised Selected Papers*. Vol. 8906. Springer. 2014, p. 333 (cit. on pp. 5, 71).

[MHR15] Enrico Mingo Hoffman and Alessio Rocchi. *OpenSoT webpage*. `https://github.com/robotology-playground/opensot`. Accessed: 2015-11-27. 2015 (cit. on p. 60).

[Mic04] Olivier Michel. „Cyberbotics Ltd. Webots TM : Professional Mobile Robot Simulation“. In: *Int. Journal of Advanced Robotic Systems* 1 (2004), pp. 39–42 (cit. on p. 6).

[Mil+11] Giulio Milighetti, Luca Vallone, and Alessandro De Luca. „Adaptive predictive gaze control of a redundant humanoid robot head“. In: *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE. 2011, pp. 3192–3198 (cit. on p. 56).

[MK89] Anthony A Maciejewski and Charles A Klein. „The Singular Value Decomposition: Computation and Applications to Robotics“. In: *Int. J. Rob. Res.* 8.6 (1989), pp. 63–79 (cit. on p. 30).

[Mou+] Thomas Moulard, Antonio El Khoury, Florent Lamiraux, et al. *Numerical Optimization for Robotics*. `http://www.roboptim.net/`. Accessed: 2015-11-10 (cit. on p. 73).

[Mou+13] Thomas Moulard, Florent Lamiraux, Karim Bouyarmane, and Eiichi Yoshida. „Roboptim: an optimization framework for robotics“. In: *Robomec*. 2013, 4p (cit. on p. 73).

[Nak+08] J. Nakanishi, R. Cory, M. Mistry, J. Peters, and S. Schaal. „Operational space control: A theoretical and empirical comparison“. In: 6 (2008), pp. 737–757 (cit. on pp. 35, 38).

[Nak90] Yoshihiko Nakamura. *Advanced Robotics: Redundancy and Optimization*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990 (cit. on pp. 30, 45).

[Nor+15a] Francesco Nori, Silvio Traversaro, Jorhabib Eljaik, et al. „iCub whole-body control through force regulation on rigid non-coplanar contacts“. In: *Frontiers in Robotics and AI* 2 (2015), p. 6 (cit. on p. 69).

[Nor+15b] Francesco Nori, Silvio Traversaro, Jorhabib Eljaik, et al. *iDynTree*. `https://github.com/robotology/idyntree`. 2015 (cit. on p. 69).

[Odh+14] Lael U Odhner, Leif P Jentoft, Mark R Claffee, et al. „A compliant, underactuated hand for robust manipulation“. In: *The International Journal of Robotics Research* (2014) (cit. on p. 13).

[OSR11] OSRF. *Open Source Robotics FOundation*. `http://osrfoundation.org/`. Accessed: 2016-1-20. 2011 (cit. on p. 64).

[Pan+12] Jia Pan, S. Chitta, and D. Manocha. „FCL: A general purpose library for collision and proximity queries“. In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. May 2012, pp. 3859–3866 (cit. on p. 43).

[Par+98] Ki Cheol Park, Pyung Hun Chang, and Seung Ho Kim. „The enhanced compact QP method for redundant manipulators using practical inequality constraints“. In: *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*. Vol. 1. ieeexplore.ieee.org, May 1998, 107–114 vol.1 (cit. on p. 42).

[Pat+10] U Pattacini, F Nori, L Natale, G Metta, and G Sandini. „An experimental evaluation of a novel minimum-jerk cartesian controller for humanoid robots“. In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. ieeexplore.ieee.org, Oct. 2010, pp. 1668–1674 (cit. on p. 33).

[Phi+11] Roland Philippsen, Luis Sentis, Oussama Khatib, and Luis Sentist. „An open source extensible software package to create whole-body compliant skills in personal mobile manipulators“. In: *IROS*. 2011, pp. 1036–1041 (cit. on p. 33).

[PK13] Florian T. Pokorny and Danica Kragic. „Classical Grasp Quality Evaluation: New Theory and Algorithms“. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Tokyo, Japan, 2013 (cit. on p. 15).

[Qui+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, et al. „ROS: an open-source Robot Operating System“. In: *ICRA Workshop on Open Source Software*. 2009 (cit. on p. 5).

[Ram+14] O. Ramos, N. Mansard, and P. Souères. „Whole-body Motion Integrating the Capture Point in the Operational Space Inverse Dynamics Control“. In: *IEEE-RAS International Conference on Humanoid Robots (Humanoid'14)*. Madrid, Spain, Nov. 2014.

[Roba] RightHand Robotics. *Reflex SF Spec Sheet*. `http://www.righthandrobotics.com/main:reflex`. Accessed: 2015-08-26 (cit. on p. 14).

[Robb] Robotiq. *3-Finger Adaptive Robot Gripper Spec Sheet*. `http://robotiq.com/products/industrial-robot-hand/`. Accessed: 2015-08-26 (cit. on p. 14).

[Roc+] Alessio Rocchi, Enrico Mingo Hoffman, Darwin G. Caldwell, and Nikos G. Tsagarakis. „OpenSoT: A Whole Body Control Framework for Hyper-Redundant Robots (Part 1)“. Submitted.

[Roc+15a] A Rocchi, E Mingo Hoffman, D G Caldwell, and N G Tsagarakis. „OpenSoT: A whole-body control library for the compliant humanoid robot COMAN“. In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. ieeexplore.ieee.org, May 2015, pp. 6248–6253 (cit. on p. 29).

[Roc+15b] Alessio Rocchi, Enrico Mingo Hoffman, Darwin G. Caldwell, and Nikos G. Tsagarakis. „OpenSoT: a Whole-Body Control Library for the Compliant Humanoid Robot COMAN“. In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE. 2015, pp. 1093–1099.

[Roc+16] Alessio Rocchi, Barrett Ames, Zhi Li, and Kris Hauser. „Stable Simulation of Underactuated Compliant Hands“. In: *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. 2016 (cit. on p. 12).

[Roca] Alessio Rocchi. *gazebo BLEM patch*. `https://bitbucket.org/arocchi/gazebo`. Accessed: 2015-08-26 (cit. on p. 24).

[Rocb] Alessio Rocchi. *robot_capsule_generator webpage*. `https://github.com/arocchi/robot_capsule_generator`. Accessed: 2015-11-27 (cit. on p. 60).

[Roh+13] Eric Rohmer, Surya P. N. Singh, and Marc Freese. „V-REP: A versatile and scalable robot simulation framework.“ In: *IROS*. IEEE, 2013, pp. 1321–1326 (cit. on p. 6).

[Rok10] Lior Rokach. „A survey of clustering algorithms“. In: *Data mining and knowledge discovery handbook*. Springer, 2010, pp. 269–298 (cit. on p. 19).

[Ros] Carlos J. Rosales. *pisa-iit-soft-hand*. `https://github.com/CentroEPiaggio/pisa-iit-soft-hand`. Accessed: 2015-08-26 (cit. on p. 24).

[Ros60] J B Rosen. „The Gradient Projection Method for Nonlinear Programming. Part I. Linear Constraints“. In: *Journal of the Society for Industrial and Applied Mathematics* 8.1 (1960), pp. 181–217 (cit. on p. 39).

[Sen+10] L Sentis, Jaeheung Park, and O Khatib. „Compliant Control of Multicontact and Center-of-Mass Behaviors in Humanoid Robots“. In: *Robotics, IEEE Transactions on* 26.3 (June 2010), pp. 483–501 (cit. on p. 33).

[Set+14] Alessandro Settimi, Corrado Pavan, Valerio Varricchio, et al. „A modular approach for remote operation of humanoid robots in search and rescue scenarios“. In: *Modelling and Simulation for Autonomous Systems: First International Workshop, MESAS 2014, Rome, Italy, May 5-6, 2014, Revised Selected Papers*. Vol. 8906. Springer, 2014, p. 192 (cit. on p. 47).

[She+11] Michael A. Sherman, Ajay Seth, and Scott L. Delp. „Simbody: multibody dynamics for biomedical research“. In: *Procedia {IUTAM}* 2.0 (2011), pp. 241 –261 (cit. on p. 7).

[Sic+08] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. 1st. Springer Publishing Company, Incorporated, 2008 (cit. on p. 46).

[SK08] Bruno Siciliano and Oussama Khatib, eds. *Springer Handbook of Robotics*. Springer, 2008 (cit. on p. 30).

[Smi+09] R Smits, H Bruyninckx, and J De Schutter. „Software support for high-level specification, execution and estimation of event-driven, constraint-based multi-sensor robot tasks“. In: *Advanced Robotics, 2009. ICAR 2009. International Conference on*. June 2009, pp. 1–6 (cit. on p. 32).

[Smi+11] Ruben Smits, H Bruyninckx, and E Aertbeliën. „KDL: Kinematics and dynamics library“. In: *Avaliable: http://www. orocos. org/kdl* (2011) (cit. on p. 33).

[Smi00] Russel Smith. *Open Dynamic Engine*. `http://www.ode.org/`. Accessed: 2016-1-20. 2000 (cit. on pp. 6, 14).

[SS91] Bruno Siciliano and Jean-Jacques E Slotine. „A general framework for managing multiple tasks in highly redundant robotic systems“. In: *Advanced Robotics, 1991.'Robots in Unstructured Environments', 91 ICAR., Fifth International Conference on*. IEEE. 1991, pp. 1211–1216 (cit. on p. 30).

[Tec13] Georgia Tech. *DART*. `http://dartsim.github.io/`. Accessed: 2016-1-20. 2013 (cit. on p. 7).

[Tik+08] V. Tikhanoff, A. Cangelosi, P. Fitzpatrick, et al. „An Open-source Simulator for Cognitive Robotics Research: The Prototype of the iCub Humanoid Robot Simulator“. In: *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems*. PerMIS '08. Gaithersburg, Maryland: ACM, 2008, pp. 57–61 (cit. on p. 64).

[Tod+12] Emanuel Todorov, Tom Erez, and Yuval Tassa. „MuJoCo: A physics engine for model-based control.“ In: *IROS*. IEEE, 2012, pp. 5026–5033 (cit. on p. 6).

[Van+12] Dominick Vanthienen, Tinne De Laet, Wilm Decré, Herman Bruyninckx, and Joris De Schutter. „Force-sensorless and bimanual human-robot comanipulation“. In: *10th IFAC Symposium on Robot Control (SYROCO)*. Vol. 10. 2012, pp. 5–7 (cit. on p. 32).

[Van+13] Dominick Vanthienen, Markus Klotzbuecher, Tinne De Laet, Joris De Schutter, and Herman Bruyninckx. „Rapid application development of constrained-based task modelling and execution using Domain Specific Languages“. In: *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Tokyo, Japan, 2013, pp. 1860–1866 (cit. on p. 32).

[WA12] Jonathan Weisz and P.K. Allen. „Pose error robust grasping from contact wrench space metrics“. In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. May 2012, pp. 557–562 (cit. on p. 15).

[Yam04] Katsu Yamane. *Simulating and generating motions of human figures*. STAR / Springer tracts in advanced robotics. evolution of the author's PhD. Berlin: Springer, 2004 (cit. on p. 42).

[Ani97] Anitescu, Mihai And Potra, Florian. „Formulating Dynamic Multi-Rigid-Body Contact Problems with Friction as Solvable Linear Complementarity Problems“. In: *Nonlinear Dynamics* 14.3 (1997), pp. 231–247 (cit. on p. 17).