The Open Stack of Tasks Library: OpenSoT

Enrico Mingo Hoffman, Arturo Laurenzi, Nikos G. Tsagarakis

▶ **To cite this version:**

Enrico Mingo Hoffman, Arturo Laurenzi, Nikos G. Tsagarakis. The Open Stack of Tasks Library: OpenSoT. 2024. hal-04621130v2

HAL Id: hal-04621130
https://hal.science/hal-04621130v2

Preprint submitted on 12 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Open Stack of Tasks Library: OpenSoT

Enrico Mingo Hoffman[1,2], Arturo Laurenzi[2], and Nikos G. Tsagarakis[2]

***Abstract*—The OpenSoT library is a state-of-the-art framework for instantaneous whole-body motion planning and control based on Quadratic Programming optimization. The library is designed to enable users to easily write and solve a variety of complex instantaneous whole-body control problems with minimal input, facilitating the addition of new tasks, constraints, and solvers. OpenSoT is designed to be real-time safe and can be conveniently interfaced with other software components such as ROS or other robotic-oriented frameworks. This paper aims to present the usage of the OpenSoT library to a large audience of researchers, engineers, and practitioners, as well as to provide insights into its software design, which has matured over nearly 10 years of development.**

***Index Terms*—Constrained reactive control, whole-body control, model-based control, software frameworks**

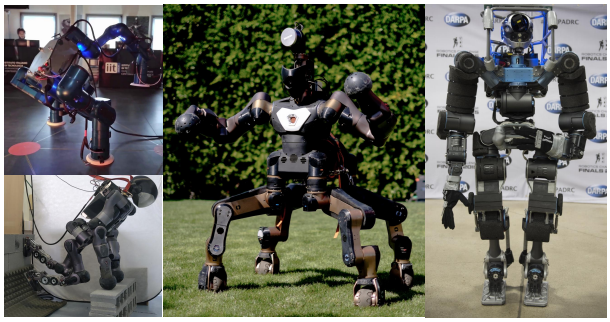

Fig. 1: Robots using OpenSoT as a whole-body motion engine include MARM in the top-left, COMAN+ in the bottom-left, CENTAURO in the center, and WALK-MAN on the right.

I. Introduction

Recent years have witnessed the emergence of sophisticated robotic systems characterized by multiple degrees of freedom (DOFs) and limbs, enabling them to navigate through and interact with their environment. For instance, mobile platforms equipped with manipulators, quadrupeds, and more recently, full humanoids, are transitioning from laboratory settings to practical applications.These systems surpass classical industrial manipulators in terms of capabilities, including a broader workspace, extended range of motions, and enhanced control of interaction forces through their whole body. However, it is widely acknowledged that these robots demand sophisticated motion and interaction controllers. These controllers must consider hardware limitations, a large number of DOFs, and multiple tasks simultaneously.

One of the most well-established control paradigms in robotics is based on computing the next control action by solving an inverse problem linearized at the robot's current state. Various real robotics systems implement control schemes following this paradigm, including resolved-rate control, closed-loop inverse kinematics, operational space control, and Cartesian impedance control, among others. These methods also manage actuation and force redundancies and adapt for use with both fixed- and floating-base robots. Optimization techniques, particularly Quadratic Programming (QP), are now the standard approach for efficiently solving these types of control problems with the possibility to include linear constraints, e.g. joint limits or collision avoidance. The development of these methods is particularly active in the context of whole-body control where multiple tasks and constraints need to be resolved simultaneously, for example when controlling humanoids and quadrupeds.

This article describes an open-source software library written in C++ to write instantaneous whole-body control problems and resolve them using optimization, designed for research, educational, and industrial applications, named *Open Stack of Tasks:* OpenSoT. OpenSoT aims at different audiences: motion planning and control researchers, robotics educators, practitioners, and end users in the robotics industry. Within the robotics community, showcasing the superiority of a new task, constraint, solver, or formulation, over existing ones, can be a daunting task. It requires a considerable investment of time and effort for a researcher to implement these components and conduct a thorough comparison with the existing state-of-the-art. Furthermore, the whole-body algorithms should effectively and efficiently address motion control problems for systems with numerous DOFs in real-time. To address this challenge, we developed OpenSoT to simplify this process and facilitate the exploration of innovative ideas. The division between abstract base classes, which solely define the interface, and derived classes responsible for implementing the specified functionality, contributes to a clearer comprehension of general concepts in whole-body planning and control.

OpenSoT is designed to be practical and applicable in real-world scenarios, and deployable in many real robotic systems, e.g. see Figure 1. Another crucial requirement is the seamless integration of OpenSoT with other software components on a robot, including high-level planning, perception, kinematics and dynamics modeling, and low-level control. This paper aims to introduce the software's evolution, illuminate key lessons learned throughout its development, and underscore its enduring significance in current robotic applications.

[1] Université de Lorraine, CNRS, Inria LARSEN Team, LORIA, France, 615 rue du Jardin Botanique, 54600 Villers les Nancy, France. E-mail: enrico.mingo-hoffman@inria.fr

[2] Humanoid and Human Centered Mechatronics (HHCM) Lab, Istituto Italiano di Tecnologia (IIT), Via Morego 30, 16163 Genova, Italy. E-mail: {arturo.laurenzi, nikos.tsagarakis}@iit.it

This work was granted by the European Union's Horizon Europe Program under Grant No. 101070596 (euROBIN) and by the French National Research Agency (ANR) under the project ANR-24-CE33-0753-01 (MeRLin).

II. State of the Art

Researchers have dedicated considerable effort to developing algorithms and software for instantaneous whole-body motion planning and control. The primary objectives of these software frameworks are twofold: first, to simplify the formulation of optimization problems, like those in (5) or (7) in the Appendix; second, to enable efficient solving of such problems. The first goal is typically addressed by segregating each task and constraint into atomic entities that can be flexibly combined to construct the optimization problem. This optimization problem is then tackled by a solver, which is often partitioned into a *front-end*, responsible for generating elements (matrices and vectors), and a *back-end*, that is, an off-the-shelf optimization library that uses these elements to solve the problem.

A pioneering work in this direction was the *Stack of Tasks* [1] (SoT), which allowed the formulation and resolution of complex Inverse Kinematics (IK) problems with hard priorities in humanoid robots, eventually extended to Inverse Dynamics (ID) [2]. In the SoT, each task is represented as an entity class loaded through a plugin within a scripting language, enabling the construction of a *stack* defining the problem to be solved, where each level of the stack corresponds to a priority level. As the reader may infer, OpenSoT draws significant inspiration from this foundational work. Similar software frameworks to the SoT were developed within several laboratories. For instance, the Optimization-Based Controllers for Robotics Applications (OCRA) [3], developed within the YARP community, for the humanoid robot iCub. Another example is the `ihmc-whole-body-controller` [4], developed for the DARPA Robotics Challenge (DRC) and based on Java. Other notable frameworks include Drake [5], ControlIt! [6], TSID [7], iTaSC [8], `mc_rt` [9], and Placo[1]. Some of these frameworks rely on a specific formulation of the instantaneous whole-body control problem (see for example [7]), assume a particular resolution algorithm (see for example [6]), or depend on specific model libraries (see for example [8]). Such design choices make them difficult to extend and may restrict users from testing different options for their problems. Furthermore, many of these software frameworks aim to function as comprehensive packages for robotics, (see for example [9]), including a list of functionalities that may not be strictly required by users who simply want to deploy whole-body controllers on existing low-level frameworks.

OpenSoT was specifically developed to focus solely on generic formulations and resolution of whole-body control problems, with a particular emphasis on *developer* and *user* experience. Thanks to OpenSoT abstraction layers and tools, a developer can easily integrate novel tasks, constraints, and solvers, thereby extending the library with new functionalities. Meanwhile, a *user* can formulate novel stacks using the existing OpenSoT components or compare different solver methods. Furthermore, OpenSoT is designed to be integrable within other low-level control frameworks. All these characteristics make OpenSoT unique in the panorama of whole-body controller libraries.

[1]https://placo.readthedocs.io/

III. OpenSoT: Overview and Main Components

In contrast to other existing whole-body control software libraries, OpenSoT does not incorporate a representation of the robot model and avoids assuming a specific controller formulation or priority resolution strategy. The advantage of this minimalist approach is that it enables the design of a library suitable for the generic creation and resolution of hierarchical QP and Linear Programming (LP) problems subject to linear constraints. For instance, beyond the conventional formulations of whole-body controllers, OpenSoT can be utilized to address problems like contact force distribution or floating-base estimation.

The OpenSoT framework builts upon the *hierarchical paradigm*: initially, an external model of the robot is updated, allowing the computation of kinematics and dynamics quantities necessary for various *tasks* and *constraints*. Subsequently, tasks and constraints are updated to compute all the matrices and vectors required to formulate and solve the optimization problem using a dedicated *Solver*. While the model can be updated using measured or derived quantities, each task and constraint contains a reference to the model to retrieve all the necessary matrices and vectors. Additionally, each task and constraint independently manages various settings, such as gains, parameters, and high-level references. Tasks and constraints are organized into a *Stack*, providing a convenient means to specify types and levels of priorities.

We will now examine the fundamental concepts associated with OpenSoT's component (see Fig. 2). Throughout the following sections, references to the robot model will be made. However, it's crucial to note that, as introduced earlier, the OpenSoT API doesn't encompass classes for representing robots. This design choice offers the advantage of creating a library that can function independently and seamlessly integrate with various generic model libraries, such as the Rigid Body Dynamics Library (RBDL), Mujoco, or Pinocchio. For tasks and constraints implemented in the library that utilize the robot's model, we depend on a generic `ModelInterface` class, developed within the XBotCore software suite, which can be easily specialized using the aforementioned model libraries. The development of the OpenSoT library commenced in 2015 as the motion engine for the COMAN and WALK-MAN robots, participating in the DARPA Robotics Challenge (DRC) Finals.The initial version of OpenSoT was closely tied to the YARP robotics framework, utilizing `yarp::sig` matrices and vectors, and the iDynTree model library. Its functionality was limited to resolving Inverse Kinematics and Differential Inverse Kinematics problems. Following the DRC, the library underwent significant restructuring, transitioning to the `Eigen` library for all linear algebra computations, and adding further capabilities.

A. Tasks

OpenSoT employs a different expression for the cost function compared to similar software frameworks:

$$\|\mathbf{A}_\mathrm{i}\mathbf{x} - \mathbf{b}_\mathrm{i}\|_{\mathbf{W}_\mathrm{i}} + \mathbf{c}_\mathrm{i}^T\mathbf{x}, \tag{1}$$

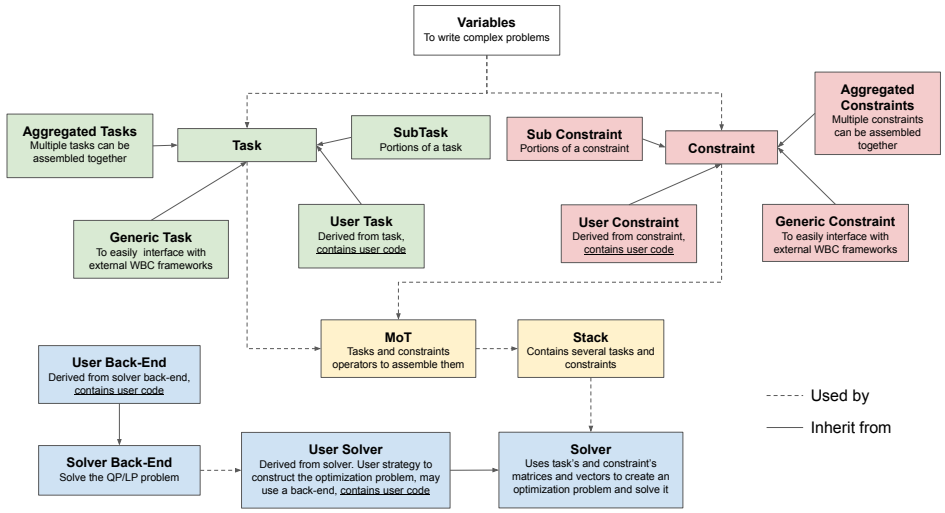

Fig. 2: Main software components in OpenSoT.

where $\mathbf{c}_i \in \mathbb{R}^n$, allowing the specification of both least-squares ($\mathbf{c}_i = \mathbf{0}$) and linear problems ($\mathbf{A}_i = \mathbf{0}$), e.g., for Lasso regression. A task object inherits from the base class `OpenSoT::Task`, where the method `update()` must be implemented to assign the matrix $\mathbf{A}$ and vectors $\mathbf{b}$ and $\mathbf{c}$, at every control loop. As mentioned earlier, this assignment is typically done by requesting kinematics and dynamics quantities from a reference to the model, which is updated outside the task. The weight matrix $\mathbf{W}$ can be set using a dedicated method exposed to users.

The `GenericTask` class provides users with methods to assign all the internal matrices and vectors, eliminating the need to derive from the base class. For this reason, the `GenericTask` is a convenient way to easily interface with libraries for prototyping reasons. To exclude joints that belong to a kinematic chain inside a task, it is possible to use the `applyActiveJointsMask()` method, which selectively sets columns in the $\mathbf{A}$ matrix of the task to $\mathbf{0}$. Finally, a task can be activated or deactivated using the `setActive()` method. When a task is not active, its $\mathbf{A}$ matrix is set to $\mathbf{0}$.

A *SubTask* comprises a specific number of rows from a task and is constructed from a task object along with the rows of interest. The `SubTask` class facilitates the selection of adjacent and non-adjacent rows from a task by extracting sub-matrices from $\mathbf{A}$ and $\mathbf{W}$, along with a sub-vector from $\mathbf{b}$. Note that SubTasks do not make use of the $\mathbf{c}$ vector. A SubTask holds a reference to the task that was used to instantiate it; therefore, the original task is still used to handle settings and references. As a result, any modifications applied to the original task are reflected in the SubTask, and vice versa. The referenced task is also updated when the `update()` method is invoked on the SubTask. A common application of a SubTask is to focus on a specific portion of a task. For instance, in a Cartesian task, a SubTask permits isolating the positional component while disregarding the orientation.

A library of ready-to-use tasks is available in OpenSoT, some of which reported in reported in Table I. Tasks with "$*$" are mostly used as equality constraints.

TABLE I: Implemented tasks in OpenSoT

Task	Formulation
Cartesian	Velocity/Acceleration
Postural	Velocity/Acceleration
CoM	Velocity/Acceleration
Angular Momentum	Velocity/Acceleration
Gaze	Velocity
Pure Rolling*	Velocity
Floating-Base Dynamics*	Acceleration + Force
Min Effort	Velocity
Manipulability	Velocity

B. Constraints

A constraint object inherits from the base class `OpenSoT::Constraint`, where the method `update()` must be implemented to assign the matrix $\mathbf{C}$, and the vectors $\mathbf{c}_l$ and $\mathbf{c}_u$, for the lower and upper bounds respectively. As for the Subtask, a *SubConstraint* comprises a specific number of rows from a constraint, and the `GenericConstraint` permits assigning internal matrices and vectors without the need to derive from the base `Constraint` class.

A library of ready-to-use constraints is also available in OpenSoT, with the most significant ones reported in Table II In particular, joint position, velocity, and acceleration limits are implemented using exponential control barrier functions and invariance. Friction cones use pyramidal approximation, and normal torque limits implement bounds on the contact normal torque.

TABLE II: Implemented constraints in OpenSoT

Constraint	Formulation	Type
Joint Pos./Vel./Acc. Limits	Velocity/Acceleration	Inequality
Torque Limits	Acceleration + Force	Inequality
CoP Limits	Wrench	Inequality
Friction Cone Limits	Force	Inequality
Normal Torque Limits	Wrench	Inequality
Collision Avoidance	Velocity	Inequality
OmniWheels	Velocity	Equality

C. Variables

In OpenSoT, the matrix $\mathbf{A}$ and the vectors $\mathbf{b}$ and $\mathbf{c}$ completely define a task, and similarly, the matrix $\mathbf{C}$ and the vectors $\mathbf{c}_\text{l}$ and $\mathbf{c}_\text{u}$ completely define a constraint. When formulating the optimization problem using these matrices, the *optimization variables are implicit* and depend on how these matrices and vectors are written. In other words, we do not explicitly write the variables of the problem; instead, we infer them from the structure of these matrices and vectors.

For example, in Differential Inverse Kinematics (DIK) (see the problem (5) in the Appendix), the generic Cartesian task is defined by the task matrix $\mathbf{A} = \mathbf{J}$, and the variables are the joint velocities $\mathbf{x} = \dot{\mathbf{q}}$. In ID problems (e.g. see problem (7) in the Appendix), the generic Cartesian task is defined by the task matrix $\mathbf{A} = [\mathbf{J} \quad \mathbf{0}]$, whose size depends on the number of contacts, in fact the variables of the problem are $\mathbf{x} = \begin{bmatrix}\dot{\boldsymbol{\nu}}^T & \mathbf{f}^T\end{bmatrix}^T$. From a developer's point of view, this is not convenient because, depending on the number of contact forces, the size of the task matrix will vary, requiring more code to handle a variable that is not used in that particular task. This also complicates the process of reusing the task, particularly in scenarios like differential inverse kinematics at the acceleration level, where contact forces and dynamics are not present. Ideally, a developer should be able to implement the task irrespective of other variables, focusing solely on joint accelerations. To address this issue, OpenSoT introduces the concept of *explicit variables* through the `OpenSoT::AffineHelper` class. An *affine* variable is defined as:

$$\mathbf{y} = \mathbf{M}\mathbf{x} + \mathbf{p} \tag{2}$$

and can be used to define (or extract) explicit variables from $\mathbf{x}$. For example, considering the the variables (see again problem (7) in the Appendix):

$$\begin{aligned}\dot{\boldsymbol{\nu}} &= \mathbf{M}_{\dot{\boldsymbol{\nu}}}\mathbf{x} \\ \mathbf{f} &= \mathbf{M}_{\mathbf{f}}\mathbf{x}\end{aligned} \tag{3}$$

with $\mathbf{M}_{\dot{\boldsymbol{\nu}}} = [\mathbf{I}_{n+6\times n+6} \quad \mathbf{0}_{n+6\times 3c}]$ and $\mathbf{M}_{\mathbf{f}} = [\mathbf{0}_{3c\times n+6} \quad \mathbf{I}_{3c\times 3c}]$. In this way is possible to write tasks and constraints independently from the definition of the optimization variables. If we consider again the Cartesian task in acceleration, the task matrix will be $\mathbf{A} = \mathbf{J}\mathbf{M}_{\dot{\boldsymbol{\nu}}}$.

The `AffineHelper` class exposes an easy-to-use API to create the vector of variables $\mathbf{x}$, compute the matrices to extract these variables, and compose variables into more complex ones. The `VariableVector` type allows stacking multiple variables together using the `emplace_back(var_name, var_size)` method. It is then possible to retrieve the variable by name using the `getVariable(var_name)` method along with the associated $\mathbf{M}$ matrix and $\mathbf{p}$ vector.

Some tasks in OpenSoT can be constructed using a variable, for example, all the tasks formulated in joint acceleration or contact forces, to be easy to combine. Utilities to transform tasks and constraints that do not make use of variables are provided, allowing the inclusion of implicit tasks and constraints in explicit stacks.

D. Stack

In OpenSoT a Stack consists of one or more tasks and constraints and their relations. Tasks and constraints can be *aggergated* together through opportune stacking of the associated matrices and vectors. Task aggregation permits easy implementation of *soft priority* strategies between tasks by setting relative weights. Notice that such operation can be seen from the point of view of the cost function as a *sum* of tasks. OpenSoT uses a Domain Specific Language called *Math of Tasks* [10] which permits to easy specify various types of relations between tasks and constraints objects. For example, the sum of multiple tasks is done by the "+" (sum) operator, while relative weights can be set by using the "$*$" (product) operator. A SubTask can be specified inside a stack using the "`%`" (modulo) operator. Finally, constraints can be inserted into a stack using the "<<" (left shift) operator. Note that this operator can also be used with tasks to create equality constraints. Applying these operators creates a stack object, implemented through the `OpenSoT::AutoStack` class. An autostack carries pointers to the associated tasks and constraints, therefore the `update()` method updates the autostack including the internal tasks and constraints.

E. Solvers

Solvers are responsible for resolving QP problems (see problem (4) in the Appendix), possibly implementing a strategy to handle *hard priorities* between tasks. A user solver is derived from the base class `OpenSoT::Solver`, where the user is required to implement the method `solve()`.

Solvers typically consist of two components: a *Front-end* and a *Back-end*. The front-end utilizes the stack to construct the matrices and vectors that will be utilized by the back-end to solve the problem. This framework can be applied to various techniques implemented in the front-end, while the back-ends can be reused, thereby promoting reusability. For instance, the inequality Hierarchical QP (iHQP) solver adheres to this framework. In the iHQP solver, the front-end calculates the Hessian and constraints and sets up a series of QP problems corresponding to the number of hard priorities specified in the stack, which are sequentially addressed. At each priority level, the front-end incorporates an additional constraint considering the solution from the preceding solved QP. Each QP is tackled by a dedicated back-end, implemented by a dedicated solver.

Back-ends can be developed by utilizing the base class `OpenSoT::BackEnd`. When creating a back-end, the methods `initProblem()` and `solve()` need to be implemented. Additional methods are mandatory: `setOptions()` and `getOptions()` to transmit and retrieve options to and from the solvers, respectively, and the method `getObjective()` to retrieve the residual from the optimization. Solvers that deviate from this pattern can be directly implemented using the `Solver` base class. Table III lists the solvers already integrated into OpenSoT. Specifically, we interface with the HCOD implementation available in the `soth` package. While Table IV lists the available back-ends in OpenSoT. It's worth noting that, unlike many similar software packages, we also provide LP and Mixed Integer Programming

TABLE III: Implemented solvers in OpenSoT

Solver	Constraints Type	Use Back-end
eHQP [11]	equality only	×
iHQP [12]	equality/inequality	✓
nHQP [13]	equality/inequality	✓
HCOD [14]	equality/inequality	×

(MIP) solvers, expanding the range of applications for these formulations in whole-body control problems, as well as facilitating their integration with QP formulations.

TABLE IV: Implemented back-ends in OpenSoT

Solver	Type
qpOASES [15]	QP/LP
OSQP [16]	QP/LP
proxQP [17]	QP/LP
qpSWIFT [18]	QP
eiQuadProg [19]	QP
GLPK [20]	LP/MIP

Figure 3 reports benchmark results on an IK problem cast on a 29 DOFs floating-base humanoid robot. We considered 4 possible tasks varying the number of hard priorities (from single layer to 4 priority levels) including joint position and velocity limits and contact constraints. The tasks include the positioning of the Center of Mass (CoM), left and right hand poses, and a joint level posture. We randomly selected 30 IK problems, each involving a transition from an initial configuration to a goal configuration defined by a specific Cartesian pose. Each IK problem is solved using the iHQP and nHQP front-end solvers, using the back-ends: OSQP, qpOASES, eiQuadProg, proxQP, and qpSWIFT. We compared the average time taken to solve a single instance of a hierarchical problem[2], including the time needed to compute the Hessian and gradient, as well as to fill the solver matrices. Additionally, we calculated the success rate (SR), defined as the number of times the back-end successfully found a solution within certain conditions, divided by the total number of runs.

To confirm that the back-end successfully performs the task, we consider the task error norm to be less than or equal to $1e^{-3}$ within the specified maximum number of iterations, set to 1000. Local minima as well as near-singular configurations may reduce the SR. For robots with multiple DOFs, simple constrained IK problems can be solved fast, within $1\ ms$ and $10\ ms$. The success rate is for most of the cases between 80% and 100%. Notably, in the case of the iHQP, the time taken to resolve multiple layers does not change significantly when increasing the number of layers to values typically used in complex robotics systems (4-5 layers). Notice that these times do not take into account the time needed to update the model and stack. This duration can vary depending on the model library utilized and the implementation of various tasks and constraints. The tests were conducted without fully leveraging the extensive capabilities of the examined QP solvers, both in terms of tunable options and optimal implementation.[3].For example, some of the QP solvers only offer a sparse interface, necessitating the conversion of dense matrices, which are typical of the problems in the whole-body framework, into sparse format. Consequently, the obtained results may be affected by suboptimal option tuning and less-than-optimal implementation. For these reasons, these results are provided solely to offer a preliminary understanding of the performance in solving specific control problems using OpenSoT.

[2]On an AMD® Ryzen 9 4900HS with 32 GiB of RAM

[3]An accurate comparison of QP solvers can be found at https://github.com/qpsolvers/qpbenchmark

IV. Main Extensions and Integration with other Frameworks

The OpenSoT control library can be integrated with other frameworks to enhance its usability and capabilities, as well as facilitate the development of new tasks or constraints. Specifically, we will here discuss four of the major extensions of OpenSoT.

A. CartesI/O & ROS

OpenSoT doesn't prescribe a specific structure for controllers but offers components for crafting controllers with an intuitive programmatic interface. However, there are specific scenarios where the controller's structure remains consistent, and the only variables are the stack and/or solver utilized. For instance, in IK, the controller's framework involves initializing the reference to the current robot state, solving the optimization problem, and integrating to calculate the subsequent reference. *CartesI/O*[4] is a high-level framework designed for two primary objectives: firstly, to provide user-friendly high-level interfaces for tasks, constraints, and solvers, and secondly, to offer readily configurable controllers.

The core concept is to furnish base classes for tasks and constraints, necessitating the implementation of an *interface* class and a *computation* class. In particular, interface classes are ROS-based, while computation classes utilize OpenSoT. Consequently, ROS topics and services are exposed for each task and constraint, facilitating easy parameter tuning and reference setting through interactive markers, among other functionalities. Each component functions as a plugin that can be loaded with a YAML file, wherein initial parameter values, stack structure, solver types, and other pertinent information are specified. This YAML file, also known as a `.stack` file, forms the basis of a controller template, automatically configured to the user's requirements. This approach enables users to construct, configure, and interact with controllers without the necessity of low-level coding. Additionally, CartesI/O provides Python bindings and integrates the Reflexxes motion library.

B. Visp

The Visual Servoing Platform (Visp) is a well-known framework in robotics dedicated to visual servoing[5].Visp is used together with OpenSoT to implement a velocity-based visual servoing task, `opensot_visual_servoing`, enabling the tracking of desired visual features using image feedback. In particular, Visp data structures are used to handle visual features and to compute interaction matrices

[4]https://advrhumanoids.github.io/CartesianInterface/

[5]https://visp.inria.fr/

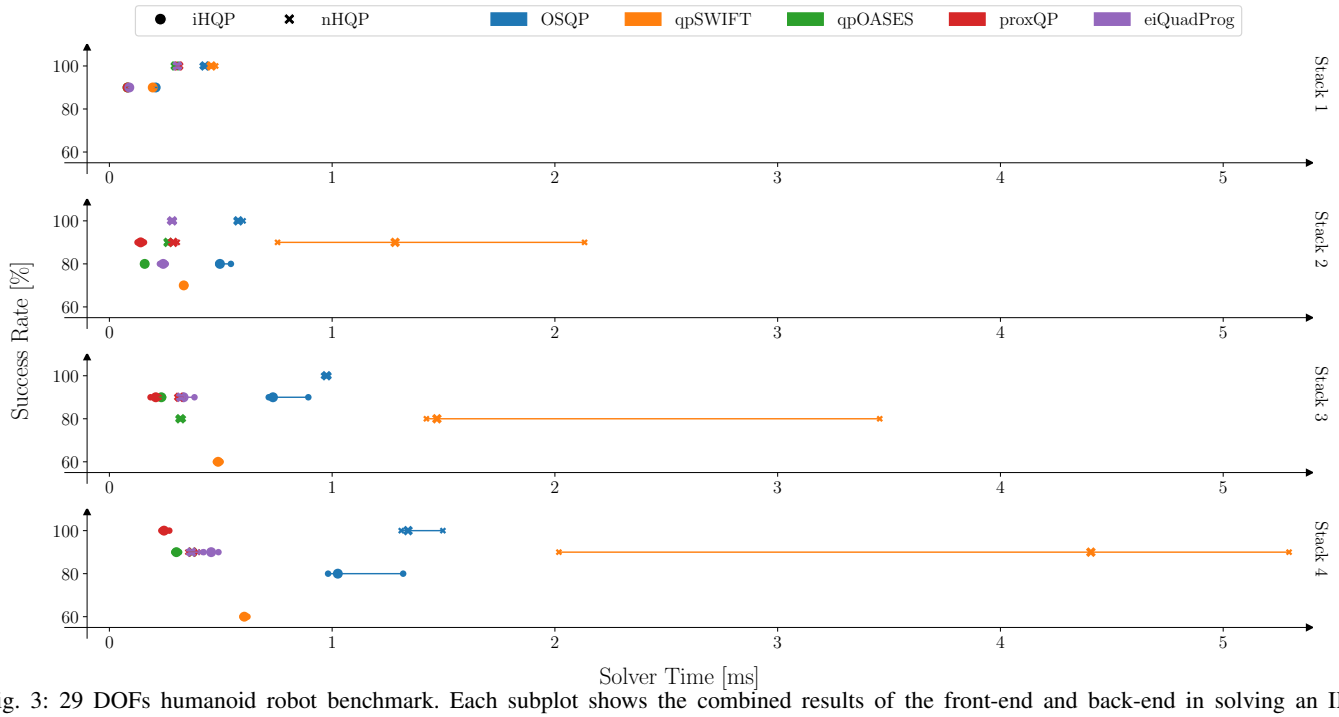

Fig. 3: 29 DOFs humanoid robot benchmark. Each subplot shows the combined results of the front-end and back-end in solving an IK problem across a range from single to four priority levels. The left ordinate axis displays the Success Rate, while in the right is reported the associated priority level in increasing order. The abscissa axis represents time. For each combination of front-end and back-end, the median and Interquartile Range (IQR) are plotted.

for mapping visual features into end-effector velocities. The `tasks::velocity::VisualServoing` task contains a `tasks::velocity::Cartesian` task object responsible for computing Jacobians related to the specified camera frame. Measured visual features are used as feedback and can be provided to the task via the `setFeatures()` method at every control loop. Visual features are computed from an RGB image using a Python script, which then transmits the results through a ROS topic to a CartesI/O interface designed for the `VisualServoing` task.

C. HPP-FCL

HPP-FCL is a well-known library utilized for efficiently computing distances between shapes[6]. This library enables self-collision and object-collision avoidance inequality constraints in OpenSoT. Self-collision avoidance is supported for any type of collision meshes. Through CartesI/O, the `PlanningScene` from ROS is leveraged to seamlessly insert or remove shapes and point clouds into the constraint.

D. OMPL

The Open Motion Planning Library (OMPL) is a comprehensive collection of state-of-the-art sampling-based motion planning algorithms[7]. Within OMPL, OpenSoT is utilized to project random state configurations for floating-base systems onto the manifold defined by the contacts through IK, implemented inside `ompl::base::Constraint`. This projection is performed while simultaneously considering secondary tasks and constraints in various forms, such as stability checks, collisions, and generalized preferred postures.

[6]https://github.com/humanoid-path-planner/hpp-fcl
[7]https://ompl.kavrakilab.org/

V. An Example

In this section, we now describe an example of the usage of the OpenSoT library to create and solve an ID stack for the humanoid robot TALOS. Our example will be written using the Python bindings API. First, we create a model object

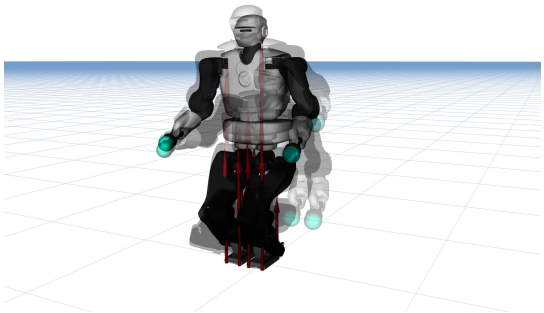

Fig. 4: Open loop whole-body inverse dynamics performed on TALOS model.

starting from a URDF description of the robot:

```
from xbot2_interface import pyxbot2_interface as xbi

model = xbi.ModelInterface2(urdf, model_type="pin")
```

The second parameter permits specifying the model library to be used inside the `ModelInterface`, i.e. Pinocchio. Once again, is worth noticing that, as reported in Section III, OpenSoT does not rely on any particular model library. The available tasks and constraints are built upon the

`ModelInterface` library which can be implemented using different model libraries.

Second, we declare the variables we will use, in particular, generalized accelerations and contact forces for each contact point defined in the `contacts` list:

```
import pyopensot as pysot

vars_vec = dict()
vars_vec["qddot"] = model.nv

for contact in contacts:
    vars_vec[contact] = 3
vars = pysot.OptvarHelper(vars_vec)
```

Third, we declare the tasks and constraints that will compose the stack:

```
from pyopensot.tasks.acceleration import *
from pyopensot.constraints.force import *

qddot = vars.getVariable("qddot")

LS = Cartesian("LS", model, "LS", "world", qddot)
...

com = CoM(model, qddot)

postural = Postural(model, qddot);
...

f_vars = list()
for contact in contacts:
    f_vars.append(vars.getVariable(contact))

df = DynamicFeasibility("floating_base_dynamics",
    model, qddot, f_vars, contacts)

LFULFc = FrictionCone(contacts[0],
    vars.getVariable(contacts[0]), model, mu)
...
```

Once the tasks and the constraints are declared, we can compose them in a stack, for example, considering a single-level stack with soft priorities:

```
stack = LS + RS + 1e-3*postural + base%[3,4,5] + ...

stack<<floating_base<<v_lims<<j_lims<<LFULFc<<...
```

notice that we are setting a relative weight of $1e^{-3}$ for the `postural` task and we are considering the orientation part for the `base` task using the operator "%" that creates a subtask from the original task using the provided list of indices, while constraints are inserted using "<<".

Finally, we instantiate the solver that will compute the optimal generalized accelerations and contact forces:

```
solver = pysot.iHQP(stack)
```

The control loop consists of the following:

```
while(control_on)
{
    model.setJointPosition(q);
    model.setJointVelocity(qdot);
    model.update();

    com.setReference(com_p_ref, com_v_ref);
    ...
    autostack.update();

    // Solve, retrieve solution and compute ID
    x = solver.solve(x)

    qddot_val = vars.getVariable("qddot").getValue(x)
    f_vals = list()
    for contact in contacts:
        f_vals.append(
            vars.getVariable(contact).getValue(x))

    tau = ID(qddot_val, f_vals, contacts);
    ...
}
```

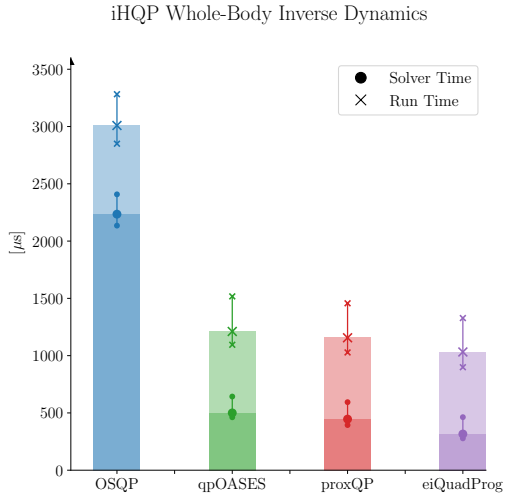

Fig. 5: Comparison between different back-ends running the whole-body inverse dynamics on the TALOS robot using the iHQP front-end. The run time considers the solver time, the update of the model and the stack, and a small overhead due to the communication to ROS provided by the CartesI/O framework.

notice that any reference or parameter variation in tasks or constraints must be done *before* the autostack update to be actualized. The `getValue()` method is used to retrieve the variable values from the solution vector `x`.

The final stack is composed of Cartesian tasks for the arm's and leg's end-effectors, CoM XY positions, centroidal angular momentum regulation to zero, orientation of the base, a postural for the upper body, and regularization of contact forces and joint accelerations. The constraints are the floating-base dynamics, joint torques, velocity and position limits, force limits, pyramidal friction cones, and the unilaterality of contact forces. The problem has a total of 60 variables, 24 contact forces and 36 generalized accelerations, and 172 constraints. The stack has been also implemented through a CartesI/O stack file permitting the exploitation of the Python CartesI/O API and ROS for setting references to tasks, and RVIZ for visualization. For the sake of simplicity, we implemented an open-loop scheme where optimal generalized accelerations are integrated twice and fed back to the model.

To solve the optimization problem we use the iHQP front-end. We command the right arm to follow a linear path involving also a movement of the entire body, as shown in Figure 4. We compared different back-end solvers, as shown in Figure 5. In Figure 6 are reported the computed torques and in Figure 7 the related contact forces.

Fig. 6: Computed torques.

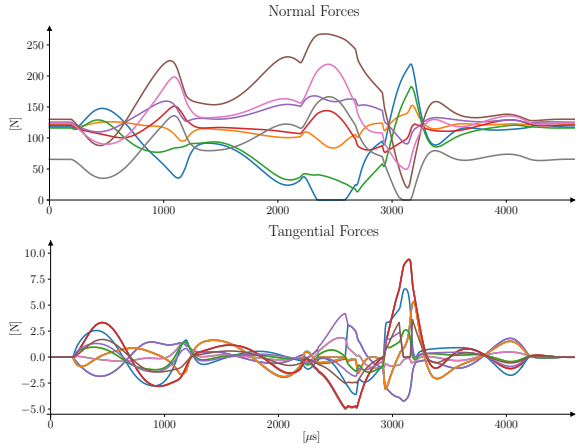

Fig. 7: Contact forces.

VI. Conclusion and Future Works

This paper described OpenSoT, an open-source library for reactive whole-body control on generic robotic systems. The main components of the library are discussed, including the OpenSoT API, and its integration with other software libraries and robotics frameworks. OpenSoT is adopted in various robotics platforms, including manipulators, humanoids, and quadrupedal robots. Its integration with CartesI/O provides an introduction to the complexity of whole-body control: without writing any code, users can construct whole-body problems by combining ready-to-use tasks and constraints written for different formulations, and solve such problems using different solvers, vary parameters, and perform extensive benchmarking experiments.

Through ongoing development in our group and contributions from others, we expect OpenSoT to become a valuable tool for researchers, robotics industry users, and students. We envision utilizing OpenSoT in various scenarios except from traditional whole-body control. It can be leveraged for modeling and analyzing the design of complex robotics systems, conducting preliminary studies, and offering valuable insights. We have already discussed its integration with OMPL for manifold projection, considering priorities. Another potential application of OpenSoT could involve state estimation, such as floating-base estimation. Lastly, we believe that it could play a role in Reinforcement Learning by reducing the size of the input of the policy through Cartesian mapping into generalized coordinates.

References

[1] 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*, 2009, pp. 1–6.

[2] O. E. Ramos, N. Mansard, O. Stasse, C. Benazeth, S. Hak, and L. Saab, "Dancing humanoid robots: Systematic use of osid to compute dynamically consistent movements following a motion capture pattern," *IEEE Robotics & Automation Magazine*, vol. 22, no. 4, pp. 16–26, 2015.

[3] J. G. Eljaik, R. Lober, A. Hoarau, and V. Padois, "Optimization-based controllers for robotics applications (ocra): The case of icub's whole-body control," *Frontiers in Robotics and AI*, vol. 5, p. 24, 2018.

[4] T. Koolen, S. Bertrand, G. Thomas, T. De Boer, T. Wu, J. Smith, J. Englsberger, and J. Pratt, "Design of a momentum-based control framework and application to the humanoid robot atlas," *International Journal of Humanoid Robotics*, vol. 13, 03 2016.

[5] R. Tedrake, "the drake development team," *Drake: A planning, control, and analysis toolbox for nonlinear dynamical systems*, vol. 5, p. 8, 2016.

[6] C.-L. Fok, G. Johnson, J. D. Yamokoski, A. Mok, and L. Sentis, "Controlit!—a software framework for whole-body operational space control," *International Journal of Humanoid Robotics*, vol. 13, no. 01, 2016.

[7] A. Del Prete, N. Mansard, O. E. Ramos, O. Stasse, and F. Nori, "Implementing torque control with high-ratio gear boxes and without joint-torque sensors," *International Journal of Humanoid Robotics*, vol. 13, no. 01, 2016.

[8] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *The International Journal of Robotics Research*, vol. 26, no. 5, pp. 433–455, 2007.

[9] A. Bolotnikova, P. Gergondet, A. Tanguy, S. Courtois, and A. Kheddar, "Task-space control interface for softbank humanoid robots and its human-robot interaction applications," in *IEEE/SICE International Symposium on System Integration (SII)*, 2021, pp. 560–565.

[10] E. M. Hoffman and N. G. Tsagarakis, "The math of tasks: A domain specific language for constraint-based task specification," *International Journal of Humanoid Robotics*, vol. 18, no. 03, 2021.

[11] F. Flacco and A. De Luca, "Discrete-time redundancy resolution at the velocity level with acceleration/torque optimization properties," *Robotics and Autonomous Systems*, vol. 70, pp. 191–201, 2015.

[12] O. Kanoun, F. Lamiraux, and P.-B. Wieber, "Kinematic control of redundant manipulators: Generalizing the task-priority framework to inequality task," *IEEE Transactions on Robotics*, vol. 27, no. 4, pp. 785–792, 2011.

[13] M. de Lasa and A. Hertzmann, "Prioritized optimization for task-space control," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009, pp. 5755–5762.

[14] A. Escande, N. Mansard, and P.-B. Wieber, "Hierarchical quadratic programming: Fast online humanoid-robot motion generation," *The International Journal of Robotics Research*, vol. 33, no. 7, pp. 1006–1028, 2014.

[15] H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl, "qpOASES: a parametric active-set algorithm for quadratic programming." *Math. Program. Comput.*, vol. 6, no. 4, pp. 327–363, 2014.

[16] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: an operator splitting solver for quadratic programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020.

[17] A. Bambade, S. El-Kazdadi, A. Taylor, and J. Carpentier, "Prox-qp: Yet another quadratic programming solver for robotics and beyond," in *Robotics: Science and Systems*, 2022.

[18] A. G. Pandala, Y. Ding, and H.-W. Park, "qpswift: A real-time sparse quadratic program solver for robotic applications," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3355–3362, 2019.

[19] D. Goldfarb and A. Idnani, "A numerically stable dual method for solving strictly convex quadratic programs," *Mathematical programming*, vol. 27, no. 1, pp. 1–33, 1983.

[20] "Gnu linear programming kit." [Online]. Available: http://www.gnu.org/software/glpk/glpk.html

APPENDIX A
BACKGROUND

We address control problems involving the *linear mapping* of quantities from operational space, such as Cartesian space, to generalized coordinates. These problems comprise various atomic elements that can be categorized into two main groups: *tasks* and *constraints*. Generally, a task can be accurately mapped into generalized coordinates when constraints are absent. Tasks can be mathematically formalized as linear least-squares pieces of a cost function to be minimized. Each task can be characterized by a task matrix $\mathbf{A}_\mathrm{i} \in \mathbb{R}^{m\times n}$, a task vector $\mathbf{b}_\mathrm{i} \in \mathbb{R}^m$, and a task weight matrix $\mathbf{W}_\mathrm{i} \in \mathbb{R}^{m\times m}$. Constraints are in the form of linear equalities and/or inequalities characterized by a constraint matrix $\mathbf{C} \in \mathbb{R}^{l\times n}$ and lower and upper constraint vectors, respectively $\mathbf{l}, \mathbf{u} \in \mathbb{R}^l$:

$$\begin{aligned} &\min_{\mathbf{x}} \sum_{\mathrm{i}=0}^{\mathrm{N}} \|\mathbf{A}_\mathrm{i}\mathbf{x} - \mathbf{b}_\mathrm{i}\|_{\mathbf{W}_\mathrm{i}} + \epsilon\|\mathbf{x}\| \\ &s.t. \quad \mathbf{l} \leq \mathbf{C}\mathbf{x} \leq \mathbf{u}, \end{aligned} \tag{4}$$

with $\mathbf{x} \in \mathbb{R}^n$ the vector of unknowns, e.g. generalized coordinates, and m the task size. The second term in the cost function in (4) is referred to as Tikhonov regularization. Its purpose is to regularize ill-posed problems that may arise during the computation of the Lagrangian function associated with the optimization problem. Without this regularization, the Hessian matrix $\mathbf{H} = \mathbf{A}^T\mathbf{W}\mathbf{A} \in \mathbb{R}^{n\times n}$ would lack invertibility to solve the Karush-Kuhn-Tucker (KKT) conditions, as $\mathrm{rank}(\mathbf{A}) = \mathrm{rank}(\mathbf{H}) = m$, when $m < n$. This particular condition is quite common when mapping a Cartesian task in a redundant robot.

When multiple tasks need to be mapped together, or if the generalized coordinates do not provide enough DOFs, *priorities* need to be established between them. Priorities can be divided into two types: *Soft* and *Hard*. Soft priorities can be mathematically formalized as weighted sums in the cost function, where high weights are assigned to the most important tasks, as in (4). Hard priorities can be mathematically formalized as low-priority tasks resolved in the null-space of high-priority ones, implementable in various forms. The distinction between hard and soft priorities lies in the fact that, in soft priorities, all tasks are concurrently minimized (in the least-square sense), and their optimality is contingent upon the relative weights assigned to each task, leading to average-like solutions when tasks conflict. In contrast, hard priorities involve minimizing secondary tasks *without* compromising the optimality of the primary task. In robotic systems with many DOFs, normally hard and soft priorities are mixed. Constraints are usually hard unless slack variables are used. Therefore, when two constraints conflict, the QP problem defined in (4) becomes *infeasible*. Two major families of mapping problems in robotics use the QP problem in (4): *differential inverse kinematics* and *inverse dynamics*.

A. Differential Inverse Kinematics

In Differential Inverse Kinematics (DIK) *instantaneous* Operational Space velocities are mapped into joint space velocities. The QP problem in the DIK case with multiple soft tasks is formulated as follows:

$$\begin{aligned} \dot{\mathbf{q}}^* = &\operatorname*{argmin}_{\dot{\mathbf{q}}} \sum_{\mathrm{i}=0}^{\mathrm{N}} \|\mathbf{J}(\mathbf{q})_\mathrm{i}\dot{\mathbf{q}} - \mathbf{v}_\mathrm{i}\|_{\mathbf{W}_\mathrm{i}} + \epsilon\|\dot{\mathbf{q}}\| \\ &s.t. \quad \mathbf{l} \leq \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} \leq \mathbf{u}. \end{aligned} \tag{5}$$

First-order integration of $\dot{\mathbf{q}}^*$ permits to solve the non-linear Inverse Kinematics (IK) problem for the ith task given by:

$$\mathbf{T}_\mathrm{i,d} = \mathbf{f}_\mathrm{i}(\mathbf{q}), \tag{6}$$

where $\mathbf{T}_\mathrm{i,d}$ is a desired Cartesian pose and $\mathbf{f}_\mathrm{i}(\cdot)$ representing the Forward Kinematic (FK) function for the ith task. The actual and desired poses are used to compute an appropriate error, which is then employed as a Cartesian velocity reference in the task. This error exponentially tends to zero based on a positive definite proportional gain.

B. Inverse Dynamics

Similarly, it is possible to formulate the QP problem in terms of generalized acceleration variables and subsequently use the optimized generalized accelerations to solve the IK problem through second-order integration. Another application of the optimized joint accelerations is the computation of joint torques associated with the motion.

An interesting case is the floating-base ID where generalized accelerations $\dot{\boldsymbol{\nu}}$ and contact forces $\mathbf{f}$ are computed simultaneously within the QP optimization:

$$\begin{aligned} &\min_{\dot{\boldsymbol{\nu}},\mathbf{f}} \sum_{\mathrm{i}=0}^{\mathrm{N}} \|\mathbf{J}_\mathrm{i}(\mathbf{q})\dot{\boldsymbol{\nu}} + \dot{\mathbf{J}}_\mathrm{i}(\mathbf{q},\boldsymbol{\nu})\boldsymbol{\nu} - \mathbf{a}_\mathrm{i}\|_{\mathbf{W}_\mathrm{i}} + \epsilon_{\dot{\boldsymbol{\nu}}}\|\dot{\boldsymbol{\nu}}\| + \epsilon_\mathbf{f}\|\mathbf{f}\| \\ &s.t. \quad \mathbf{M}_b(\mathbf{q})\dot{\boldsymbol{\nu}} + \mathbf{h}_b(\mathbf{q},\boldsymbol{\nu})\boldsymbol{\nu} = \mathbf{J}_{c,b}^T(\mathbf{q})\mathbf{f} \\ &\qquad \mathbf{J}_c(\mathbf{q})\dot{\boldsymbol{\nu}} + \dot{\mathbf{J}}_c(\mathbf{q},\boldsymbol{\nu})\boldsymbol{\nu} = \mathbf{0} \\ &\qquad \mathbf{l} \leq \mathbf{C}(\mathbf{q},\boldsymbol{\nu}) \begin{bmatrix} \dot{\boldsymbol{\nu}} \\ \mathbf{f} \end{bmatrix} \leq \mathbf{u}, \end{aligned} \tag{7}$$

with $\mathbf{q} \in \mathbb{R}^n + SO(3)$, $\boldsymbol{\nu} \in \mathbb{R}^{n+6}$, $\mathbf{J}_c(\mathbf{q}) \in \mathbb{R}^{3c\times n+6}$ the stack of the Jacobians of the contacts and $\mathbf{J}_{c,b}(\mathbf{q}) \in \mathbb{R}^{3c\times 6}$ its floating-base part, $\mathbf{M}_b(\mathbf{q}) \in \mathbb{R}^{6\times n+6}, \mathbf{h}_b(\mathbf{q},\boldsymbol{\nu})\boldsymbol{\nu} \in \mathbb{R}^6$ the floating-base dynamics and non-linear terms, respectively, and $\mathbf{f} \in \mathbb{R}^{3c}$ where c are the number of contacts. The first and second constraints are the *underactuation* and *contact* constraints, respectively. The underactuation constraint relates contact forces to generalized accelerations, while the contact constraint ensures that contacts do not move. Finally, the third generic constraint represents constraints applied to generalized accelerations and contact forces, such as joint limits, unilateral forces, and friction cones.

The optimized joint accelerations and contact forces can be used to compute the actuated joint torques $\boldsymbol{\tau} \in \mathbb{R}^n$:

$$\mathbf{M}_j(\mathbf{q})\dot{\boldsymbol{\nu}} + \mathbf{h}_j(\mathbf{q},\boldsymbol{\nu}) = \boldsymbol{\tau} + \mathbf{J}_{c,j}^T(\mathbf{q})\mathbf{f}, \tag{8}$$

with $\mathbf{J}_{c,j}(\mathbf{q}) \in \mathbb{R}^{3c\times n}$ the joint space part of the stacked contact Jacobians, and $\mathbf{M}_j(\mathbf{q}) \in \mathbb{R}^{n\times n}$ and $\mathbf{h}_j(\mathbf{q},\boldsymbol{\nu}) \in \mathbb{R}^n$ respectively the inertia matrix and non-linear terms in joint space.