A Whole-Body Software Abstraction layer for Control Design of free-floating Mechanical Systems

Francesco Romano Silvio Traversaro Daniele Pucci Francesco Nori
iCub Facility Department, Istituto Italiano di Tecnologia, Genoa, 16163, Italy

Abstract—In this paper, we propose a software abstraction layer to simplify the design and synthesis of whole-body controllers without making any preliminary assumptions on the control law to be implemented. The main advantage of the proposed library is the decoupling of the control software from implementation details, which are related to the robotic platform. Furthermore, the resulting code is more clean and concise than ad-hoc code, as it focuses only on the implementation of the control law. In addition, we present a reference implementation of the abstraction layer together with a Simulink® interface to provide support for Model-Driven based development. We also show the implementation of a simple proportional-derivative plus gravity compensation control together with a more complex momentum-based bipedal balance controller. Before concluding the paper we present an honest view of the lessons we have learnt during the development of the library, i.e. we report both good and bad design choices that have had a strong influence on the usability and maintainability of the library.

Index Terms—whole-body control; software architecture; free-floating mechanical systems.

1 INTRODUCTION

NOWADAYS, robotics is moving from the original industrial context to more human-like environments. Foreseen applications involve robots with augmented autonomy and physical mobility. Within this novel context, physical interaction influences stability and balance. Consequently, the requirements and tasks that we expect from some platforms are changing as well. Instead of precise positioning tasks confined in cages in industrial assemblies, robots are foreseen to help in everyday-life tasks such as cleaning houses or elderly assistance.

The increase in complexity of robotic systems demands an increase in complexity of the corresponding control software. While ad-hoc solutions can be easy to implement, it is important to consider scalability, flexibility and portability of the developed software. The possibility to use the same software to control more than one platform can be of enormous importance in simplifying the testing, tuning, and deployment of the same controller on different robots.

Whole-body control has received an increased attention by the robotics community because of the possibility it offers to accomplish tasks coordination and to fully take advantage of the robots dynamics in presence of contacts. Indeed, the possibility to specify multiple objectives, even conflicting, at the same time opens the possibility to properly exploit robots for complex scenarios. In particular, citing the definition from the RAS Technical Committee [1] “Whole-Body Control aims to i) define a small set of simple, low-dimensional rules (e.g., equilibrium, self collision avoidance, etc.) ii) that are sufficient to guarantee the correct execution of any single task, whenever feasible [...], and of simultaneous multiple tasks [...], iii) exploiting the full capabilities of the entire body of redundant, floating-based robots in compliant multi-contact interaction with the environment”.

In the context of whole-body control the Task function approach [2] has been successfully used as a method to specify control objectives at a high level. In particular, control objectives are represented as $n$-dimensional continuous output functions, called tasks, to be regulated to zero. All the tasks, together with possible constraints are then transformed into a constrained optimisation problem.

We propose a software abstraction layer specifically targeted at simplifying the design of whole-body controllers for mechanical systems. We deal with the control problem from a general perspective, without limiting the user to the use of a task-based approach or other specific control frameworks. Indeed, when we consider a generic control system, we usually

Regular paper – Manuscript received July 15, 2017; revised November 15, 2017. Digital Object Identifier: 10.6092/JOSER_2017_08_01_p89

• This work was supported by the FP7 EU project CoDyCo (No. 600716 ICT 2011.2.1 Cognitive Systems and Robotics).

• Authors retain copyright to their papers and grant JOSER unlimited rights to publish the paper electronically and in hard copy. Use of the article is permitted as long as the author(s) and the journal are properly acknowledged.

identify three main building blocks:

- Plant model. If we consider a model-based control system, in this block the information about the plant model given the current plant state is computed.
- Feedback from the plant. This usually implies the possibility to obtain the current state of the controlled plant.
- Actuation. The control system must interact with the plant.

Any control-oriented software library must provide the above features to be of any use. Given the complexity of robotic systems it can be difficult, time consuming and error prone to write the controller directly in a low-level programming language such as C++. Nevertheless the control library must be efficient as it is usually required to have fast control loops. The aforementioned requirements serve as motivation for a model-driven development approach in such control libraries.

In this paper we propose a software abstraction layer which is responsible for decoupling the control software from i) the actual interface used to obtain the state feedback; ii) the actual interface used to command the actuation; iii) the dynamic software library used to represent the robot dynamical model. Furthermore the proposed library is scalable and easily portable to other robots or different configurations.

This paper is an extension of the work originally presented in [3]. In addition to what has been previously presented, in this paper we provide a detailed description of the current C++ reference implementation together with the Simulink® interface for enabling model-driven design of the robot controllers. The proposed library is the foundation of all the controllers we have been developing in the last four years. Given the experience we have acquired in this period we decide to report the lessons we have learned during the development and use of the library. We think it might be a valuable contribution for the community.

The paper is structured as follows. Section 2 lists the software middleware currently used in the robotics community to provide a level of abstraction from the actual hardware platform and the software for whole-body control. Section 3 introduces the mathematical formulation of the dynamics of mechanical systems commonly used in the whole-body control formulation. Section 4 describes the architecture of the proposed whole-body abstraction library and its key elements. A specific implementation is instead presented in Section 5. Section 6 reports the mathematical formulation, the implementation and the results for two controllers implemented with the proposed library. Design choices, together with their pros and cons, are presented in Section 7. Finally Section 8 draws the conclusions.

2 Related Software

The implementation of a whole-body controller on a real robot typically involves interacting with several software components, such as the interface to the robot, and a kinematics and dynamics library. In this section we briefly review these kind of software components together with some task-based whole-body control libraries and model-driven robotics control libraries.

2.1 Robotic Middleware and Interfaces

Robotic software architectures are, typically, distributed architectures. In fact, several capabilities of the robot or the software are distributed to different nodes in the architecture: for example, joint level controllers are typically implemented in embedded low-level micro-controllers, while high level control, estimation or perception algorithms are typically deployed to relatively high performance machines.

Custom communication protocols over buses, e.g. CAN [4] or EtherCAT [5], [6] are used for exposing robot functionalities, such as joint level control or sensor readings, between high performance machines and the embedded motor controllers. Between high performance machines instead, the same functionalities are typically exposed using classical distributed communication paradigms such as Publisher/Subscriber paradigm as in the ROS middleware [7] or Observer paradigm as in the YARP middleware [8]. These communication paradigms are typically implemented over several possible transport layers, such as the TCP and UDP network protocols, or shared memory, if the components communicating run on the same physical machine. However, the use of complex middleware in code typically requires non-trivial boilerplate code for each piece of software that needs to communicate with the robot in some way, and it hardcodes in the control software the dependency on the middleware. To encapsulate the communication aspects, robot interfaces libraries are typically developed. In some cases, these libraries can easily switch between communicating with the robot over a middleware such as YARP/ROS or by communicating directly with low-level micro-controllers using the internal bus of the robot. Examples of such interfaces are the YARP Devices Drivers [9, Section 4], ros_control interfaces [10] or XBotCore interfaces [11].

2.2 Multibody Dynamics Libraries

Several Multibody Dynamics libraries suitable to be used for model-based whole-body control exists. Different type of libraries include: i) code generators, which emit code for computing specified quantities, e.g. Jacobians, Mass Matrix, etc., on specific robot models such as SD/Fast [12], Robo-tran [13] or RobCoGen [14], [15], ii) multibody simulators, which offer support for whole-body control such as [16] or [17], or iii) libraries specifically design for control and analysis such as [18], [19] and [20]. Typically these libraries do not depend on a specific middleware to work, but, for using them in whole-body control, it is often necessary to write glue code for handle any impedance mismatch between the multibody dynamics library and the control interface of the
robot. A possible mismatch consists of the fact that the Rigid
Body Dynamics library and the middleware use two different
serialisation to represent the joints of the robot, or that the
control interface of the robot includes joints (such as the
fingers) that are not of particular interest for the level of detail
required by a whole-body controller.

2.3 Task-Based Control Libraries
On top of robotics middleware and Multibody Dynamics
libraries, there are several libraries that implement Task-based
control, such as the Stack of Task (SoT) [21], OpenSoT [22],
ControllIt! [23] and the Instantaneous Task Specification using
Constraints (iTaSC) [24]. The above software allows the user
to specify the objectives and constraints but they solve the
control problem internally. On one hand, this greatly simplifies
the control problem, as only the high-level objectives have to
be specified. On the other hand, these libraries force the user
to use a specific task-based approach to obtain the control
current, such as the Puma560
robotics is the Peter Corke’s Robotic toolbox [25]. This tool-
box focuses on serial-link manipulators, such as the Puma560
software or computing
concepts. As a consequence, the user
to focus on the
domain model

2.4 Model-driven Robotics Control Libraries
Model-driven engineering is a methodology that allows the
user to focus on the domain model instead of focusing on
software or computing

3 Dynamics of a Mechanical System
This section introduces the mathematical formulation com-
monly used in the robotics literature to describe the dynamics
of mechanical systems, such as robots. Because a precise
formulation of the mathematical problem is out of the scope of
the present paper, we refer the interested reader to books on
dynamics of mechanical systems [26], [27], [28] and control
systems [29], [30] for further readings.

3.1 Notation
Throughout the section we will use the following definitions:
- $I$ denotes an inertial frame, with its z axis pointing
against the gravity.
- $I_n \in \mathbb{R}^{n \times n}$ is the identity matrix of size $n$; $0_{m \times n} \in \mathbb{R}^{m \times n}$ is the zero matrix of size $m \times n$ and $0_n = 0_{n \times 1}$.
- Given two orientation frames $A$ and $B$, and vectors of
coordinates expressed in these frames, i.e. $A p , B p \in \mathbb{R}^3$, respectively, the rotation matrix $A R_B$ is such that $A p = A R_B B p + A o_B$, with $A o_B \in \mathbb{R}^3$ the origin of the frame
$B$ w.r.t. $A$.
- We denote with $S(x) \in \mathbb{R}^{3 \times 3}$ the skew-symmetric matrix
such that $S(x) y = x \times y$, where $\times$ denotes the cross
product operator in $\mathbb{R}^3$.
- Given a function of time $f(t) \in \mathbb{R}^n$, we denote with $\dot{f}(t)$
its derivative with respect to time, i.e. $\dot{f}(t) = \frac{df(t)}{dt}$

3.2 System modelling
We assume that the mechanical model is composed of $n + 1$
rigid bodies – called links – connected by $n$ joints with one
degree of freedom each. In addition, we also assume that the
multi-body system is free floating, i.e. none of the links has an
a priori
constant pose with respect to the inertial frame. This
implies that the multi-body system possesses $n + 6$ degrees of
freedom. The configuration space of the multi-body system can
then be characterised by the position and the orientation of
a frame attached to a robot’s link – called base frame $B$ – and
the joint configurations. More precisely, the robot configuration

can be represented by the triplet
$$q = (\xi p_B , I R_B , q_j)$$

where $(\xi p_B , I R_B)$ denotes the origin and orientation of the
base frame expressed in the inertial frame, and $q_j$ denotes the
joint angles.

The velocity of the multi-body system can then be charac-
terised by the triplet
$$\nu = (\xi p_B , \xi \omega_B , q_j)$$

where $\xi \omega_B$ is the angular velocity of the base frame expressed
w.r.t. the inertial frame, i.e. $\xi R_B = S(\xi \omega_B) \xi R_B$.

We also assume that the robot is interacting with the
environment through $n_e$ distinct contacts. The application of
the Euler-Poincaré formalism [31, Ch. 13.5] to the multi-body
system yields the following equations of motion:

$$M(q) \ddot{\nu} + C(q, \nu) \nu + G(q) = B \tau + \sum_{k=1}^{n_e} J_{c_k}^T f_k$$  (1)
where $M \in \mathbb{R}^{n+6 \times n+6}$ is the mass matrix, $C \in \mathbb{R}^{n+6 \times n+6}$ is the Coriolis matrix and $G \in \mathbb{R}^{n+6}$ is the gravity term. $\tau$ are the internal actuation torques and $B$ is a selector matrix which depends on the available actuation, e.g. in case all joints are actuated, i.e. $\tau \in \mathbb{R}^{n}$, it is equal to $B = (0_{n \times 6}, 1_n)^\top$. $f_k = [F_k^\top, \mu_k^\top]^\top \in \mathbb{R}^6$, with $F_k, \mu_k \in \mathbb{R}^3$ respectively the force and corresponding moment of the force, denotes an external wrench applied by the environment on the link of the $k$-th contact. The Jacobian $J_{C_k} = J_{C_k}(q)$ is the map between the robot velocity $\nu$ and the linear and angular velocity

$$\mathcal{I}_{C_k} := (\mathcal{I}_{p_{C_k}}, \mathcal{I}_{\omega_{C_k}})$$

of the frame $C_k$, i.e.

$$\mathcal{I}_{C_k} = J_{C_k}(q)\nu.$$

## 4 Software Architecture

We propose a software abstraction layer to simplify creating whole-body controllers for highly redundant mechanical systems. Given the requirements introduced in the previous sections we highlight four main elements that must be present in the library, i.e. *Actuators*, *Sensors*, *State* and *Model*. The abstraction offered by the library allows one to also easily implement higher-level interfaces such as a Simulink® interface. Figure 1 summarises the whole software architecture.

Note that the paradigm described by this library does not assume any particular robot operating system or underlining software as this is left to the actual library implementation.

A crucial feature of the proposed abstraction layer is related to the ordering of the information provided from and to the robot. In fact, the elements that directly interface the hardware, i.e. the *Actuators*, *Sensors* and *State* have to represent the information in a robot-dependent suitable way. On the other hand, the *Model* element usually interfaces with libraries that represent the information with the formalism of Eq. (1). To further complicate the problem, the control software may want to access only a subset of the degrees of freedom modelled by the dynamics library, or provided by the robot. The whole-body abstraction library must thus orchestrate all the various elements to provide a unified interface to the control software.

We now describe in detail the role that each element has in the proposed library.

### 4.1 Actuators

The *Actuators* element abstracts the actual control of the robot motors. In particular it exposes the possible motors controllable mode, e.g. position control, velocity control and torque control just to cite the most common. Of course, it also provides the possibility to specify the references for the low level controllers.

### 4.2 Sensors

The *Sensors* element is the counterpart of the *Actuators* element. In fact, it abstracts all the sensors available on the robot, usually the readings from encoders, force/torque sensors or accelerometers and it is responsible for providing access to the latest sensor measurements.

### 4.3 State

The *State* element represents all the possible information which can be measured or estimated on the robot. This implies that *state* encompasses the information provided by the *sensor* element. Furthermore, it provides additional information which can come from estimation or filtering of the data. For example, if the robot provides only joint position measurements, e.g. coming from the joint encoders, a first and second derivative filter can provide velocity and acceleration measurements. In case this information is provided by the robot itself, no additional processing is required from the interface. It is important to notice that in both cases the control software using the abstraction library will remain exactly the same.

### 4.4 Model

The last element is the *Model* element. It abstracts the kinematic- and dynamic-related information that a controller needs for computing the control law. In general data are represented with the formalism of Eq. (1). Note that a common requirement for a control library is to control only a subset of the degrees of freedom of a robot, e.g. control only the lower body of a legged robot while walking. For this reason, the library must correctly compute the kinematics and dynamics of the whole system, while considering the possibility to expose only a subset of the quantities as requested by the control software.

## 5 Implementation

This section describes the current implementation of the whole-body abstraction library conceptually described in Section 4. The code has been implemented in C++ because of its widespread use and computational performance while remaining a relatively high-level programming language. The implementation has been divided in two libraries: the wholeBodyInterface [32], which represents the abstract concepts introduced in Section 4 and the yarpWholeBodyInterface [33] which is an implementation for YARP-powered robots. We do not provide a ROS [7] implementation, but this could be done by mainly interfacing with the ros_control library.

### 5.1 wholeBodyInterface

The wholeBodyInterface is the direct transposition in C++ of the abstract concepts described in Section 4. The four elements, i.e. *Actuators*, *Sensors*, *State* and *Model*, are represented as abstract classes.
The model element lists all the methods that a kinematic and dynamic library should provide. Note that the interface is “stateless”, i.e. the output of the methods can be generated by using only the input parameters. The other three elements, actuators, sensors and state, provide access to their functionalities by using a similar interface, i.e. all the methods take as parameter the type of control mode, sensor and state as an enumeration value. This allows the interface to remain generic with respect to the available type.

The library provides an additional class, wholeBodyInterface, that inherits from the four single abstract elements. This object allows users of the library to access the “robot” with a single object, and not by keeping reference to the four constituent elements. Figure 2 shows the UML class diagram for these classes.

In addition, the library provides utilities to identify the various degrees of freedom. This is one of the main features of the library and it has been revealed to be of paramount importance during its use. In particular, the degrees of freedom are identified by a unique string. For performance reason, the set of degrees of freedom is then accessed with numerical indexing during runtime. The functions to perform, and query, this map are provided by the library.

As it represents the coded counterpart of the abstraction library, wholeBodyInterface does not make any assumption on the underlining robot framework, or how data is organised, using only native C++ types: vectors and matrices are accessed directly by using the buffer raw pointer, e.g. double*.

5.2 yarpWholeBodyInterface

The yarpWholeBodyInterface is the actual implementation of wholeBodyInterface specifically considering YARP-powered mechanical systems [8]. Figure 3 shows the UML class diagram for the main classes of the library and their relation to the abstract wholeBodyInterface library described in Section 5.1. In the more general diagram describing the software architecture in Figure 1, the yarpWholeBodyInterface implements the blocks fully coloured in the “Implementations” layer of the architecture.

Regarding the model implementation we choose as kinematic and dynamic library the iDynTree library [34] and information about the kinematic and dynamic model can be loaded from different sources, e.g. a URDF representation. It is worth noting that, while the methods interface are stateless, the underlining library is stateful. Calls from multiple threads to the model methods must then be synchronised properly.
The actuators and sensors elements directly interact with YARP control boards for commanding (the actuators) and reading the measurements (the sensors). Because a robot possesses in general multiple control boards, these two elements are also responsible for mapping the information coming from the control boards to the degrees of freedom selected by the library user. This mapping is managed by helper classes that are not shown in Figure 3.

The state element contains a periodic thread which is responsible for retrieving measurements from the available sensors and, by using also model information, of estimating additional quantities. Additionally, a filter bank filters the measurements at specified cut-off frequencies, independent for each source type.

As it is partially shown in the UML diagram, the implementation uses YARP data structures, e.g. vectors, matrices, etc., for storing information and as parameters for the non-inherited methods.

### 5.3 Simulink Interface for Model-Driven Engineering

C++ applications can leverage the advantages of the proposed abstraction layer while keeping full control of the performance of the control software by directly using the provided C++ implementations. On the other hand, coding and testing a complex control system directly in C++ can be prohibitive. For example, even the simple task of monitoring a signal over time can be complex and requires the use of a dedicated library. The use of software to design and simulate dynamical system models greatly helps the design and synthesis of control systems. Domain-specific software for dynamical systems is a specific case of model-driven engineering [36].

We currently implemented the Simulink® interface to our proposed whole-body abstraction library, which can be found in [37]. Most of the features accessible in C++ are also accessible to Simulink® models. Furthermore, because the connection with the robot or the simulator is handled by the underlining C++ library, the Simulink® interface does not require any particular toolbox to command the robot, e.g. Simulink Coder™.

A further advantage of using Simulink® with respect to the C++ code consists in the possibility to exploit the abundance of toolboxes and MATLAB native functions out of the box.

Finally, by leveraging the state-of-the-art stiff and non-stiff integrators already available on Simulink®, the proposed interface clears the way for straightforwardly simulate a robotic system with application in both planning and control.

Figure 4 shows the UML class diagram for the main classes in the library. The architecture of the library is centered around the Block abstract class, which represents a single block. The class interface allows to manage the block life cycle, starting from configuration, to initialisation, execution and termination. The WBIBlock abstract class is a specialisation of the Block class offering utility methods to configure an wholeBodyInterface singleton object. Similarly, WBIModelBlock manages only the model element of the interface. Blocks such as MassMatrix, or YarpRead implements the final behaviour and are exposed to the higher level interface.
Fig. 3. UML class diagram for yarpWholeBodyInterface library. The classes that are not part of the library are shown in light grey. Each of the main abstract elements, i.e. model, sensors, actuators and states and the wholeBodyInterface class have their concrete implementation in the library. The yarpWholeBodyEstimator class, which was not present in the abstract library, is responsible for periodically providing to the state element, estimates and filtered version of the measured quantities. As before, this diagram only shows the main classes and methods provided by the library.

Note that all the methods in the Block hierarchy do not explicitly use Simulink® function calls but are agnostic on the actual software: only generic objects such as BlockInformation are used. Information on the specific implementation, i.e. Simulink® in this case, is limited in specific files.

All the blocks are allocated through the factory method instantiateBlockWithClassName of the Block class. In the Simulink® implementation, the block name is passed as first parameter directly from the Simulink® S-Function configuration.

6 EXPERIMENTS

This section presents the mathematical formulation and the implementation of a simple PD plus gravity compensation controller. We also briefly discuss the mathematical formulation and the results of a more complex controller, namely a momentum-based balancing controller which has been implemented with the Simulink® interface described in Section 5.3.

6.1 PD plus Gravity Compensation

6.1.1 Mathematical formulation

As a first example of dynamic robot controller we present the classic Proportional Derivative (PD) plus Gravity compensation controller.

This kind of controller has been usually applied to fully-actuated fixed-base robots. Considering the model presented in Section 3.2 this means that the base frame position and orientation are constant and known a-priori and thus they are not part of the robot state.

The control objective is the asymptotical stabilisation of a desired constant joint configuration \( q^d_j \) or equivalently the asymptotical stabilisation to zero of the error \( \tilde{q}_j := q_j - q^d_j \).

The choice of the following control action

\[
\tau = G_j(q) - K_P \tilde{q}_j - K_D \dot{q}_j
\]

where \( K_P, K_D \in \mathbb{R}^{n \times n} \) are the positive definite proportional and derivative gain matrices and \( G_j(q) = [0_{n \times 6} I_n] G(q) \), satisfy the control objective, i.e. the stabilisation to zero of \( \tilde{q}_j \), and it can be proved by Lyapunov arguments [26, Sec. 6.5.1].

6.1.2 Implementation

Given the simplicity of the controller described in Section 6.1.1 we show both the C++ code (see Code 1 and 2) and the Simulink® model diagram (see Figure 5). Note that, while the Simulink® diagram completely represents the controller, the C++ code snippet has been extracted from the main loop function, i.e. the function which runs at every iteration. How the control thread is created and managed depends on the
The Block class is the main abstract class of the library. It provides the interface that a generic block should implement in order to be integrated in an environment such as Simulink. Two additional abstract classes, i.e. WBIBlock and WBIModelBlock, specialise for supporting either the full wholeBodyInterface object or the iWholeBodyModel objects respectively. The implementation blocks inherit the correct abstract class, that is either the Block class if they do not need support for the wholeBodyInterface object, or the WBIBlock or WBIModelBlock if they need full or partial support of wholeBodyInterface. All the dependences from the specific dynamical system model software, e.g. Simulink, is isolated into the BlockInformation class hierarchy. This allows to easily port the library to different softwares such as Xcos\cite{35}.

The snippet of code in Code 1 shows how the specific YARP-based implementation is instantiated. In particular, the current implementation needs information about the URDF model representing the kinematic and dynamic information of the robot and the mapping between the model joints and the YARP control boards. This is provided by the object created at line 4 and passed to the interface constructor at line 7. Additionally, the list of controlled joints are passed to the interface at line 19, just before the interface initialisation routine is called.

Reading the code in Code 2 it is possible to observe how all the details regarding the specific robot platform are hidden by the library. The object robot, in fact, is accessed through its abstract type, as it can be also seen during its instantiation, i.e. in line 7 of Code 1. In Code 2, lines 4 - 7, the state of the robot, i.e. \((\dot{q}_j, \ddot{q}_j)\), is read. The feedforward term, corresponding to \(G(q)\) is computed at lines 10 - 14 where the last parameter is the resulting gravity compensation term. Finally the error and the feedback term necessary to implement Eq. (2) is computed in lines 17 - 22. Because we did not use any specific mathematical library, we explicitly computed the term \(K_p\dot{q}_j + K_d\ddot{q}_j\) in the for loop. Finally, at line 25 we send the torque command to the robot, which we previously setup to be controlled in torque mode.

Figure 5 shows the same code implemented directly in Simulink\textsuperscript{®}. It is evident how the block-based diagram is clearer with respect to its C++ counterpart. Furthermore, the possibility to add scopes, or dump signal variables directly into MATLAB workspace greatly increases its advantages with respect to directly coding in C++.
6.2 Momentum-based Balance Control

To show the power and flexibility of the proposed architecture we present here a second example. As before, we briefly introduce the mathematical formulation and we show the results of a momentum-based balancing controller which has been synthesised directly by using the Simulink® interface. Given the complexity of the control problem we do not report here screenshots or code snippets of the Simulink® model, but the model can be freely examined in [38].

\[ \dot{H} = \sum_{i=1}^{n_c} \text{CoM} X_i f_i + m \ddot{g}. \]

Here \( f_i \in \mathbb{R}^6 = [F_i^\top, \mu_i^\top]^\top \) is the \( i \)-th of the \( n_c \) contact wrenches, \( \text{CoM} X_i \in \mathbb{R}^{6 \times 6} \) is the matrix transforming the corresponding wrench from the application frame to a frame attached to the center of mass with the same orientation of the inertial frame \( I \), \( m \) is the robot total mass and \( \ddot{g} \in \mathbb{R}^6 \) is the 6D gravity acceleration vector. By assuming as virtual control inputs the contact wrenches

\[ f = [f_1^\top, \cdots, f_{n_c}^\top]^\top, \]

it is possible to control the robot momentum by solving the
following minimisation problem:

\[
\begin{align*}
\min_f & \quad \left\| \dot{H} - \dot{H}^* \right\|^2 \\
\text{s.t.} & \quad Af \leq b
\end{align*}
\]  

(3)

where the inequality constraint \(Af \leq b\) represents center of pressure, positivity of the normal component of the force and the linear approximation of friction cone constraints. The desired momentum rate of change is given by

\[
\dot{H}^* := \dot{H}^{ref} - K_p(H - H^{ref}) - K_i \int_0^t (H - H^{ref}) \, dt,
\]

(4)

where \(K_p, K_i \in \mathbb{R}^{6 \times 6}\) are positive definite block diagonal gain matrices. It is worth noting that the first three lines of Eq. (4) are equivalent to:

\[
\ddot{x}_{\text{CoM}} = \ddot{x}^{ref}_{\text{CoM}} - \bar{K}_p(x_{\text{CoM}} - \dot{x}^{ref}_{\text{CoM}}) - \bar{K}_i(x_{\text{CoM}} - \dot{x}^{ref}_{\text{CoM}}),
\]

(5)

where \(\bar{K}_p, \bar{K}_i \in \mathbb{R}^{3 \times 3}\) are the positive definite gain matrices associated with the linear momentum in Eq. (4). In practice, by controlling the robot momentum, we can impose a desired center of mass trajectory.

The second objective is responsible for constraining the joint variables and avoid internal divergent behaviours [39]. As before, we can specify a minimisation problem also for this second task, i.e.

\[
\begin{align*}
\min_{\tau} & \quad \left\| \tau - \psi \right\|^2 \\
\text{s.t.} & \quad Af \leq b
\end{align*}
\]

(6a)

\[
J\dot{\nu} + \dot{J}\nu = 0
\]

(6b)

\[
\psi := h_j(q, 0) - J^{(j)\top}(q_j - q_j^{ref}) - K_d q_j
\]

(6c)

\[
\text{Eq.}(6b)\text{ describes the free-floating dynamics of the mechanical system as described in Eq. (1) where we define}
\]

\[
h(q, \nu) := C(q, \nu)\nu + G(q)
\]

the nonlinear bias force, and

\[
J = [J_1^\top, \cdots, J_{n_c}^\top]
\]

the stack of the contact Jacobians. Eq.(6c) is the constraint equation describing the kinematic constraints associated with the contacts. Eq.(6d), which resembles a PD plus gravity and contact wrenches compensation, plays the role of a desired joint torque reference where \(h_j\) and \(J^{(j)}\) denotes the joint space bias term and Jacobian respectively.

We solve problems (3) and (6) by formulating a single hierarchical optimization problem:

\[
f^* = \arg\min_f \left\| \tau^*(f) \right\|^2
\]

(7)

\[
\text{s.t. } Af < b
\]

\[
\dot{H}(f) = \dot{H}^*
\]

\[
\tau^*(f) = \text{minimizer of (6)}.
\]

The joint torques commanded to the robot are finally obtained by \(\tau = \tau^*(f^*)\). It is worth noting that (3) has multiple solutions in case of more than one controlled contacts, as is the case in the present application. Different choices of contact forces result in different performances of the robot while balancing. In our control architecture, we choose the forces that minimise the internal torques. We refer the interested reader to [41] for a discussion on the choice of the contact forces.

It is possible to use the presented control framework to perform different tasks, by coordinating the momentum reference \(H^{ref} \), \(\dot{H}^{ref}\) and the joint equilibrium configuration \(q_j^{ref}\).

We use finite-state-machines (FSMs) to coordinate switching between different contact situations and to impose internal motions. Figure 6 shows a simplification of the FSM used to achieve the results shown in the next section. The state machine possesses three main states, corresponding to the contact configuration. Switching between states occurs depending on the measurements of the external contact forces. In each state, the controller is the one described by Eq. (7).

### 6.2.2 Results

The YouTube video [42] shows the robot performing complex movements by using the controller and the FSM implemented and running as a Simulink Model. By using the yarpWholeBodyInterface implementation we also leverage the capabilities of the YARP middleware to seamlessly connect to the real or simulated system. In particular the test platform is the iCub humanoid robot [43], endowed with 53 degrees of freedom, 6-axis force/torque sensors and distributed tactile skin, see Figure 7a. The robot is simulated on the Gazebo simulator [44] by means of Gazebo-YARP plugins [45]. The same demo has also been implemented on a different configuration of the iCub platform [46], see Figure 7b, and in simulation on the Walk-Man humanoid robot [47], see Figure 7c. Note that the two iCub robots have a different set of degrees of freedom, while the Walk-Man robot possessed completely different physical dimensions, e.g. iCub weights 33kg while Walk-Man weights 120kg. Thanks to the flexibility of the library, the controller code remains the same in all three scenarios.

We encourage the interested reader to test the controller on the Gazebo Simulator. Instructions on how to run the controller can be found directly in the model repository readme [38].

### 7 Lessons Learned

The proposed abstraction layer, together with the reference implementation described in Section 5 has been used successfully in the last four years. All the controller we implemented are based on this library, lately with more focus on Simulink interface.

In this section we report the lessons learnt during the development and the daily use of the proposed library. While
the abstract concepts have been found valid and useful, some implementation details need to, and are currently being, revisited.

### 7.1 Use of elementary data types
During the initial phase of the development of the library, we decided not to bind the users to any particular data type. Indeed, functions accepting matrices and vectors have been implemented as accepting generic pointers to floating point numbers, i.e. `double*`.

As it turned out, this is a bad design choice. The advantage gained in terms of generality is minimal with respect to the ambiguities introduced. Disadvantages of this choice can be listed as:

- Necessity to clearly document the input parameter in the documentation, and not in the code itself. While having a clear documentation is advisable in every software library, we should aim to self-documented code.
- Impossibility to validate the user input. As the raw pointer is a low-level construct, it does not encode enough information for checking sizes, or any other advanced check.
- Limitation in expressiveness. This may sound counterintuitive, but it is related to the previous point. Consider the example of passing a matrix as a parameter to a function. The buffer pointer does not indicate neither the sizes, nor the matrix storage ordering. Furthermore, the current choice completely rules out the possibility to use sparse matrices as they need more than one buffer to be managed.

### 7.2 Platform-related Configuration
The possibility to easily configure the library for the available sensors and robots have been deeply exploited by the developed controllers. Indeed, the possibility to run the same controller on very different humanoids platforms has been permitted by the library support to configuration.

The original implementation of the library did not provide this feature, and the biggest overhaul of the library in 2014 was devoted to this particular aspect. All the robot-specific information was moved outside of the code and brought to configuration files. These allowed not only to test the controller on different platform, but also to adapt to different situation on the same robot, e.g. a joint not working which should be removed from the list of controlled joints. It is important to note that it is not possible to change configuration at run time. This is done by design, as changing configuration is in general a costly operation that can easily break the real time requirements of any controller.

### 7.3 Compile-time dependence on the implementation
In Section 7.2 we discussed about the possibility to configure the controllers for the specific robot platform, e.g. the configuration of the joints, the type and number of sensors, etc., at configuration time.

As we are proposing an abstraction layer library, the possibility to choose a particular implementation at configuration time instead of compile time, can be beneficial. For example, this would be handy for moving the controller from a YARP-based to a ROS-based platform without recompiling the user code.

While this limitation can be minimal in C++ applications, i.e. by changing the actual object allocated in the application, it becomes a pressing issue for the Simulink(R) layer. Indeed, it is impossible to change the implementation in the Simulink(R) model. For providing a solution to this problem, the library

Fig. 7. Momentum-based balancing controller running on three different robots. (a) is the latest version of the iCub humanoid robot, (b) is the iCub Heidelberg version without head and arms, and (c) is Walk-man, a humanoid robot built for disaster-recovery scenarios.
should adopt a factory pattern to decide the particular implementation object to be created. Both configuration and the actual object creation can then be abstracted.

In addition, it is possible to further ease the interoperability of the library by providing a C-level interface instead of C++ classes. On one hand, this simplifies the possibility to use the library from other languages. On the other hand, a complete refactoring of the library structure would be needed to support a C interface.

7.4 (Ab)Use of inheritance

Inheritance is a powerful concept which is at the base of object oriented programming. Thanks to it, it is possible to implement abstraction patterns, such as the one proposed by the wholeBodyInterface library.

Nevertheless, it is quite easy to misuse this pattern, and this practice is quite diffuse in the C++ world. Inheritance, in fact, is often used as the first choice for extending functionalities, even when other design patterns should be preferred, e.g. delegation.

In the implementation of the abstraction library described in Section 5.1, we use inheritance also to “group” the four elements in a single object: the wholeBodyInterface class inherits the four classes iWholeBodyActuators, iWholeBodyState, iWholeBodySensors and iWholeBodyModel. This is a classic example of misuse of the inheritance feature. Indeed, wholeBodyInterface should use composition as its scope is to simply group the four elements. A change in this direction in the implementation implies that:

- It is not needed to implement all the inherited methods.
- This process is tedious and error prone as these methods are simply calls to the “corresponding element” method.
- More important, it is easier to instantiate wholeBodyInterface objects with partial elements, e.g. without the actuators element for an application that do not need to send commands to the low-level control loops.

7.5 Presence of threads in the library

The proposed abstraction library models as two separate entities the sensor and the state elements, as explained in Section 4. This separation translates in the current implementation with the fact that the state element possesses its own thread of execution. While this not problematic in general, we encountered several issues in the last years due to this decision.

The idea of the abstraction library is to provide an interface layer for commonly used concepts in whole-body control, thus separating the need to code directly for a specific interface. The presence of a subpart of the library which has its own thread of execution does not entirely fit in the abstraction idea. This is further complicated by the fact that the presence of multiple threads is hidden and not clearly stated neither in the documentation, nor in the code. As an example, a call to a start method would at least have made clearer the presence of the thread in state element.

An additional problem due to the presence of an hidden run-loop in the library is due to the synchronisation of the control logic to an external clock which is different from the machine hardware clock. This requirement appears when we want to synchronise the controller with a simulator, necessary for properly testing the performance of robot controllers.

A different choice would be to provide, in the abstraction library, the interface for an estimator object, describing the facilities that it should provide. It is then responsibility of the user to choose an estimator and to manage its life cycle explicitly.
7.6 Stateful vs Stateless implementation

Stateful or stateless implementations have their own advantages and disadvantages. The methods exposed by the model interface are inherently stateless as they represent “instantaneous” functions, and this is the reason why they have been defined as stateless functions. An advantage of stateless functions is that they do not necessitate of synchronising their access as they use only local variables. As a consequence, it is possible to execute multiple, and parallel, calls to the function. As an example, for computing finite differences, functions must be called at different configuration points, and this can be parallelised.

On the other hand, stateful functions must be correctly synchronised, and they might lead to limited parallelism. Nevertheless, as they possess a state, it is possible to optimise and execute certain operations only once for different function calls. Additionally, considering that in real-time code it is not advisable to perform dynamic memory allocation inside the run-loop, stateful implementation are usually the only possible way to avoid allocating memory to contain temporary variables needed during the computations.

In our reference implementation we adopted a stateful implementation because of this last motivation, i.e. the need to avoid dynamic memory allocation during the run-loop execution. Unfortunately, as the interface has been thought to be stateless, this creates confusion to the user and the behaviour has to be clearly documented.

7.7 Level of Abstraction

When we started developing the library implementation, one of the main features of this library was to provide an abstraction layer for the controllers user-code from the actual robot infrastructure. We motivated this feature with the idea of being able to seamlessly run an whole-body controller on very different platforms. As an example, it should have been possible to run a balancing controller coded by using the proposed interface on the YARP-powered iCub robot, or on the LAAS HRP-2 robot, or on other humanoid robots.

As it turned out, during the development of robots controllers in the last four years, we often resorted to dynamically downcast the abstract wholeBodyInterface object to the actual yarpWholeBodyInterface object, thus voluntarily giving away the abstraction feature. Indeed, while the theoretical mathematical formulation of whole-body controllers works regardless of the particular robot, the actual, working, implementation cannot overlook specific characteristics of the robot. For example, the possibility to specify control gains for the low level controllers, which is out of the original scope of the library, makes a difference for having a controller works in practice. Nevertheless, some of these features may be abstracted at the cost of rendering more complex the interface.

There is an additional, important, point which acts against the possibility to fully abstract the controller code. The proposed library voluntarily neglects aspects such as i) how do we represent vectors and matrices? ii) how do we run the controller on the specific platform? iii) how do we configure and communicate with the controller at run-time? When the user implements the controller, he has to do a choice for the previously mentioned details. This choice binds and limits the portability of the controller on different platforms. It seems natural then to ask if it makes sense to keep the library abstract while in the end the controller will not be able to remain abstract.

It is worth noting that in the current library implementation we are in presence of a double abstraction layer. In fact, we leverage the YARP middleware, which already provides an abstraction layer on the available hardware. The same applies if we were using the ROS middleware. If you have a direct control on the “lower” abstraction layer, i.e. YARP in our case, it might be a better choice to directly extend it for the needed features. In this respect, we extended the YARP controlboard plugins with a new ControlBoardRemapper plugin. This new controlboard device implements one of the main features advocated by this library, i.e. support for grouping and ordering of the robot degrees of freedom.

8 Conclusions

In this paper we presented a software abstraction layer to simplify the development of whole-body controllers. While there are already some whole-body control software libraries, they already define the controller structure and leave to the user only the possibility to specify objectives and constraints.

On the other hand the proposed library leaves complete freedom to the control designer by exposing all the information needed. It does not make any assumptions on the controller structure. The whole-body abstraction library presents also the following advantages:

- it decouples the writing of the controller from a particular robot implementation
- it decouples the writing of the controller from a specific dynamic library implementation
- it allows more concise and clear code as it represents uniquely the code needed to implement the mathematical formulation of the controller. All the implementation details are left to the library
- it allows to benchmark the controller on different platforms or with different implementations.

Furthermore, the possibility to expose the functionality at an higher level than C++ facilitates the writing of controllers as the results on the iCub robot clearly prove.

We voluntarily did not consider some aspects as they are out of the scope of the present contribution. Nevertheless they must be taken into account when a controller is implemented and used on the real system. In particular the following details should be considered:
• how are controllers run on the platform? Do they run as threads?
• how are controllers configured and initialised?
• how is communication with other software performed? For example, how are desired values provided to the controller, coming from a planner or higher-level control loop?

By not considering these details in the abstraction library, we render the library portable to different systems. Indeed, the actual control law is not concerned by the previously listed implementation details.

While the more complex demos have been achieved by directly executing the Simulink® model connected to the robot, we recognise the need to automatically generate self-contained C++ code. The advantage is twofold. On one side the autogenerated code is in general more optimised than the code directly executed in Simulink®, even if less optimised than ad-hoc C++ code. On the other side, this would remove the requirement of having a Simulink® installation on the computers controlling the robot.

As a last contribution of this paper, we presented the good, but especially the bad, design choices adopted during the implementation of the library and how they had influence on its development and use. We think that this contribution might be of help when refactoring the library or when creating a new library of this kind. We tried to be as much honest as possible, without saving any criticisms to the current implementation.

**REFERENCES**


[36] D. Brugali, “Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics,” *IEEE Robotics Automation Magazine*, vol. 22, no. 3, pp. 155–166, Sept 2015. 5.3


[38] ——, “Momentum-based Torque Balancing Controller,” https://go.ingl/MbW8Aq, 2016. 6.2.2


**Francesco Romano**

Francesco Romano received his bachelor and master in computer engineering (with specialisation in Robotics and Automation) with highest honours from University of Genoa in 2008 and 2011 respectively. In April 2016 he obtained a Ph.D in Robotics at the Italian Institute of Technology under the supervision of Francesco Nori with a thesis on whole-body control methods for humanoid robots. He is currently a Post Doctoral fellow at the iCub Facility department at the Italian Institute of Technology. His research interests include nonlinear control and optimal control applied to humanoid robotics and whole body motion.

**Silvio Traversaro**

Silvio Traversaro received his B.Sc in computer engineering and M.Sc in robotics engineering from University of Genoa in 2011 and 2013 respectively. He received his Ph.D. from the Italian Institute of Technology in 2017. His research interests include multi-body dynamics-based modelling, estimation and identification techniques applied to humanoid robotics and whole body motion.

**Daniele Pucci**

Daniele Pucci received the bachelor and master degrees in Control Engineering with highest honors from “Sapienza”, University of Rome, in 2007 and 2009. He also received the “Academic Excellence Award” from “Sapienza” in 2009. Then, in 2013 he received the PhD title in Information and Communication Technologies from University of Nice Sophia Antipolis, with a thesis prepared at INRIA Sophia Antipolis, France, under the supervision of Tarek Hamel, and Claude Samson. The PhD program was jointly with “Sapienza”, university of Rome, so he also received a PhD in Control Engineering from “Sapienza”. From 2013 to 2015, he has been a junior postdoc funded by CoDyCo - EU Project number: 600716. From 2015 to 2017 he has been a senior postdoc in the Dynamic Interaction Control research line of the Istituto Italiano di Tecnologia. Since August 2017, he took the lead of the Dynamic Interaction Control. His research focus is on the humanoid robot locomotion problem, with specific attention on the control and planning of nonlinear systems.

**Francesco Nori**

Francesco Nori was born in Padova in 1976. He received his D.Eng. degree (highest honors) from the University of Padova (Italy) in 2002. During the year 2002 he was a member of the UCLA Vision Lab, University of California Los Angeles. In 2003 Francesco Nori started his Ph.D. under the supervision of Prof. Ruggero Frezza at the University of Padova, Italy. In the year 2006 he moved to the University of Genova and started his PostDoc at the laboratory for integrated advanced robotics (LiraLab), beginning a fruitful collaboration with Prof. Giorgio Metta and Prof. Giulio Sandini. In 2007 Francesco Nori has moved to the Italian Institute of technology and in 2017 he joined DeepMind. Francesco is currently coordinating the H2020-EU project An.Dy (id. 731540); in the past he has been involved in two FP7-EU projects: CoDyCo as coordinator and Korobot as principal investigator.