The Compliant Joint Toolbox for MATLAB

An Introduction With Examples

© PHOTOCREDIT

By Jörn Malzahn, Wesley Roozing, and Nikos Tsagarakis

This article presents the Compliant Joint Toolbox for the modeling, simulation, and controller development of compliant robot actuators. The object-oriented toolbox is written in MATLAB/Simulink. In a few lines of code, it can batch-generate ready-to-use joint actuator model classes from multiple parameter sets, incorporating a variety of nonlinear dynamics effects.

The toolbox implements a selection of state-of-the-art torque and impedance controllers and features tools for numeric and analytic actuator analysis and comparison. This article introduces the main toolbox features, complete with copy-pastable code examples.

The Components Driving Robots

Actuators are the central components that make robots move. Novel applications for robot technology demand actuation design paradigms and techniques that are fundamentally different than those in traditional robotics. Robots are meant to physically collaborate with humans in unstructured environments, such as agile industrial production with low batch sizes and even in our everyday households. Actuators, apart from being the movers (as with conventional bulky, position-controlled industrial robots), now become central components that also make robots actually perceive interaction forces.

Recent developments in the design and control of torque-controlled actuators have already led to remarkable advancements in safety, robustness, and interaction performance for torque-controlled robots and assistive robotic devices. Still, the development of actuators for robotic devices relies largely on an engineer's intuition and experience rather than on any rigorous theory that guides the proper balancing of demands and takes into account other criteria, such as peak power, maximum load capacity, and torque and motion bandwidth or impact resilience. The literature lacks a proper explanation of the relevant requirements, along with metrics for their quantification, to guide this process. Conventional notions (such as power density, peak torque, maximum speed, and single numbers for bandwidths) are insufficient for new applications dominated by physical interaction.

The Compliant Joint Toolbox is available on Git under the GNU General Public License v3.0. The toolbox can

Digital Object Identifier 10.1109/MRA.2019.2896360
Date of publication: 8 April 2019

1070-9932/19©2019IEEE

also be inspected on Code Ocean [20]. It emerged during our work on actuator modeling, design, and control aimed at solutions for the actuator design challenges in diverse robotic applications. While the toolbox's first use was for modeling and simulation, it was truly helpful to rapidly interface with real actuator hardware for data recording, testing, debugging, and tuning of joint torque controllers. The toolbox helped us cope with a variety of actuator configurations in terms of, for example, rotor, gearbox, and torque sensor combinations, during the development of the WALK-MAN, CENTAURO, and CogIMon robots (www.walk-man.eu, www.centauro-project.eu, and https://cogimon.eu/). It supported the study of the dynamics and control of different compliant electrical actuators with integrated torque sensors across arbitrary parameter ranges to assess what would be viable actuator designs and investigate the impact of nonlinear phenomena, such as friction and ripple. The toolbox was an essential tool in the preparation of publications on modeling, observer design, torque, and impedance control. Through this process, the code has reached a certain level of maturity that permits productive use.

The Compliant Joint Toolbox is implemented in MATLAB/Simulink, a proprietary software suite for technical computing and rapid algorithm prototyping. The MATLAB language is interpreted, requires low learning effort, ships with numerous state-of-the-art algorithms and visualization tools, and so offers a short time to productivity. The Python language shares most of these features. Being nonproprietary, Python would have been our preferred choice for implementing the Compliant Joint Toolbox and so making it available to the community entirely for free. However, the crucial aspect that triggered the decision against a purely nonproprietary solution was the lack of a mature and sufficiently powerful open alternative for the features afforded by the Simulink Real-Time Toolbox. It provides the chance to quickly interface with the actuator hardware based on standard industrial protocols, such as Controller Area Network, Ethernet, and EtherCAT. This allows the rapid development, deployment, tuning, and testing of models and controllers on different actuator hardware and even with the actuator hardware in the loop. Doing so minimizes the time and effort required to port developed concepts from simulation to experiments, thus improving realism in research.

We hope the Compliant Joint Toolbox can catalyze the ongoing discussion on compliant robot actuation, support academic education in the field, and contribute to community efforts toward common notions, metrics, and benchmarks that ease torque-controlled actuation design, comparison, and selection across diverse robotic applications. Hence, it is our desire to make the Compliant Joint Toolbox public and draw community attention to it.

An Overview: The Toolbox Architecture

Figure 1 provides an illustration of the architecture of the toolbox. The Compliant Joint Toolbox adopts a variant of the factory design pattern to create joint model classes with different dynamics and parameter sets. The `jointBuilder` class forms the basis of this creational design. It utilizes the abstract `genericJoint` class as an interface and derives new actuator model classes from this. The Compliant Joint Toolbox notion of an actuator model comprises a mathematical structure (i.e., mathematical formulas) and a set of values for the parameters present in the mathematical structure. As a consequence, two models with different mathematical structures can have the same values for their common parameters. Conversely, two models with the same mathematical structure but different parameter values are considered two different models. Following this notion, the user specifies the desired joint model or an entire set of different models, in terms of physical parameter sets and the structure comprising the linear and nonlinear dynamics to be incorporated. To do so, the user instantiates a `jointBuilder` and calls the `buildJoint` method, which constructs the joint model class according to this specification.

A separate module, `datasheetGenerator`, automates datasheet compilation for the implemented joint models, providing an immediate picture of the

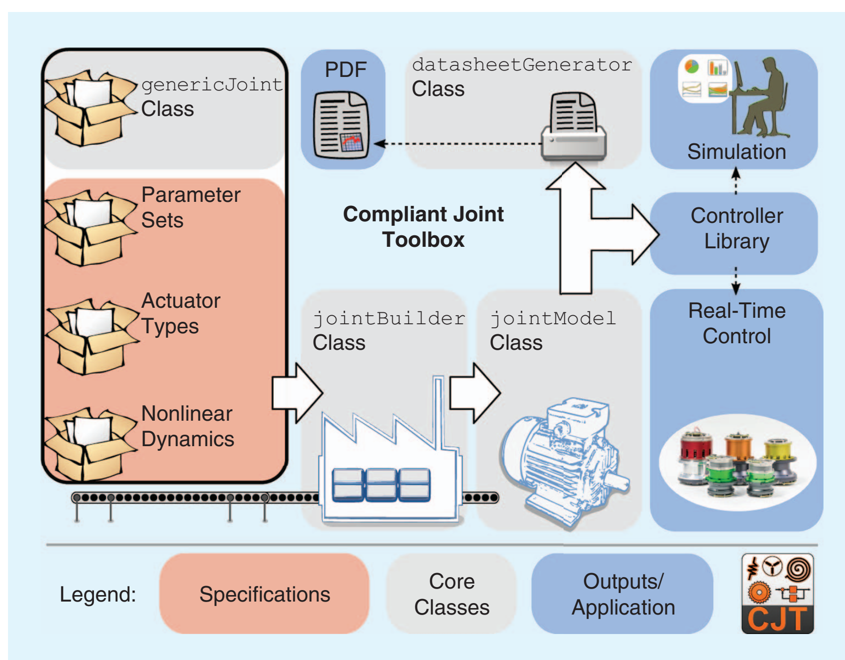


Figure 1. The Compliant Joint Toolbox architecture, adopting a variant of the abstract factory creational design pattern.

joint's capabilities using both established and novel actuator performance metrics. Controllers provided with the toolbox make use of the model objects for simulation or even real-time hardware control.

The remainder of this section briefly introduces the individual modules and how to obtain and set up the toolbox.

The `genericJoint` Class

The `genericJoint` class is the virtual class serving as a mold for actuator model classes created by `jointBuilder`. It lists joint parameters, assigns default values to them, and defines generic methods to access joint properties. A parameter set example is provided in the "Model Parameters" section. The `genericJoint` methods offer conversion between discrete- and continuous-time representations of state-space models and transfer functions; transform reflected inertia and friction parameters between the motor and load side; compute motor characteristics, such as the torque–speed slope, no-load speed, and stall torque; and convert between numeric and symbolic model representations (the Symbolic Math Toolbox is required).

The `jointBuilder` Class

The number and complexity of the relevant dynamic effects vary from actuator to actuator and depend on the user's design or control objective. Furthermore, the same principal dynamics lead to diverse results when parameterized differently. As indicated in Figure 1, the `jointBuilder` class assembles parameter sets, actuator model types, and nonlinear dynamics into `jointModel` class definitions derived from `genericJoint` classes and stores them in separate m-files. By default, the Compliant Joint Toolbox locates parameter sets in the directory `~toolboxroot/param`, while the actuator model types and nonlinear terms are prototyped in `~toolboxroot/model/`. Created m-files are stored in the build directory specified via the `jointBuilder.buildDir` property.

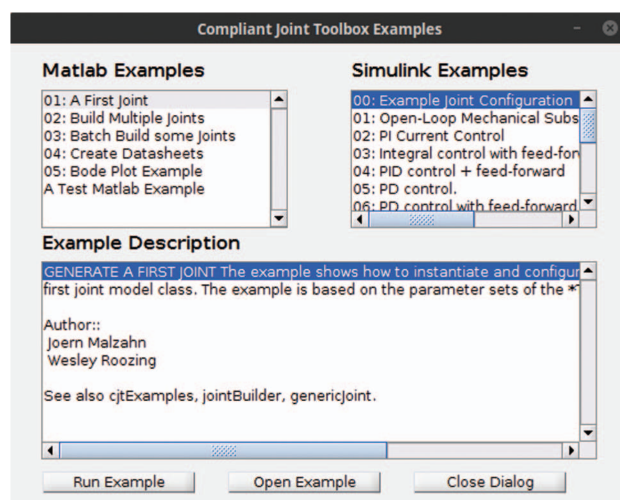


Figure 2. The browser example (`cjtExamples.m`) allows browsing through available MATLAB and Simulink offerings to inspect or directly run them.

The use of `jointBuilder` is exemplified in the "Model Generation" section.

The `datasheetGenerator` Class

The `datasheetGenerator` produces datasheet files for actuator models created by the `jointBuilder`. To this end, it relies on a LaTeX environment installed on the user machine. The generated datasheets make up a table listing the parameters of the actuator along with detailed descriptions of each parameter. Figures display the actuator characteristics, such as the torque–speed curve, efficiency curve, torque bandwidth, and thermal operation characteristics. These plots are detailed in the "Joint Model Analysis" section. The basic use of `datasheetGenerator` is demonstrated in the "Analysis Plots and Datasheet Generation" section, and an example datasheet is provided. Apart from the generation of fully formatted datasheets, the visualization functionality needed to produce embedded graphs for custom analysis is available to the user through a public method interface.

Simulation and Control

A result of research efforts on the modeling, design, and control of torque-controllable robot actuators, the Compliant Joint Toolbox features a Simulink block library implementing state-of-the-art torque controllers. The available controllers, detailed in the "Controllers" section, are implemented in discrete-time masked Simulink blocks and use previously generated joint model classes for configuration. In this way, the Simulink code generation and real-time control features (the Mathworks Real-Time Workshop and/or Simulink Coder toolboxes are required) can be exploited with the Compliant Joint Toolbox to rapidly prototype a control system for an experimental hardware setup. An example is described in the "Interfacing With Hardware" section.

Getting Started

To get started, all that is necessary is to obtain the toolbox code from its GIT repository at https://github.com/geez0x1/CompliantJointToolbox. To add the Compliant Joint Toolbox to a MATLAB search path, change the current MATLAB working directory to the toolbox directory, and run `setCJTPaths.m`. You can begin from the available examples by using the browser example displayed in Figure 2 or by following the Quickstart guide, which is provided online [21] as a short version of the complete toolbox documentation [22]. Alternatively, the toolbox can be inspected and run completely contained on Code Ocean [20].

Generating Joint Models

The Compliant Joint Toolbox comprises linear models of both the mechanical actuator subsystem and the electrical actuator subsystem as well as a number of parasitic and nonlinear effects. This section details how to provide the parameters for such dynamic effects and how to generate model classes from them. The following sections provide a number of code examples, which can be found in `Take_a_Tour.m`.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Generic Model Implementation

The linear electrical and linear mechanical subsystem models (Table 1) <**AU: Please check placement of table reference.**> of a compliant electrical actuator form the core of the Compliant Joint Toolbox. Nonlinear terms use the states of these subsystems and modulate their input–output behavior, which allows the capture of a broad range of practically relevant nonlinear dynamic effects.

The Electrical Subsystem

The most common electrical drive in torque-controlled robotic actuators is the brushless dc motor, which can be operated such that the actual three-phase motor dynamics are well described by a single-phase approximation. The governing parameters are the electrical resistance and inductance. In torque-controlled electrical actuators, the inductance is typically designed to be low. As a consequence, the electrical time constant becomes very small ($\approx 10^{-6}$ s) compared to the mechanical time constant ($\approx 10^{-3}$ s). Unless it is specifically intended to analyze the current control performance or its implications for higher-level controllers, the electrical dynamics can be neglected with respect to the mechanical time constant. This substantially shortens the simulation time. Hence, a static model is used by default in building actuator models. The "Model Generation" section describes how to switch to a dynamic model.

The Mechanical Subsystem

The mechanical subsystem is modeled as a chain of rotating masses interconnected via massless spring-damper elements, as depicted in Figure 3. The electrical drive rotor has an inertia I_m, which experiences a damping d_m with respect to ground. The gearbox contributes an inertia I_g and can be compliant with a linear stiffness k_g and internal damping d_{mg}. Gear friction with respect to ground is captured by d_g. The second elastic element is represented by a massless torsional spring with linear stiffness k_b and internal material damping d_{gl}. Finally, the rotary inertia I_l models the load with frictional damping d_l. The motor, gearbox, and load angles are denoted by q_m, q_g, and q_l, respectively. The torques acting on the motor, gearbox, and load are τ_m, τ_g, and τ_l, respectively.

Deriving the linear equations of motion for this three-mass system from first principles is straightforward and can even be found in many textbooks on control or structural dynamics, such as [1]. The Compliant Joint Toolbox features several variants of this general model structure, such as a rigid gearbox, complete rigidity (a single moving mass with friction), and fixed-output configurations. In the latter, the load motion is defined by an external source, effectively allowing the connection of the actuator model to the complex articulated robot dynamics. Load motion can be zero to emulate a locked actuator output or, equivalently, infinitely high load inertia. This last scenario is often used for torque controller design and analysis [2]–[4].

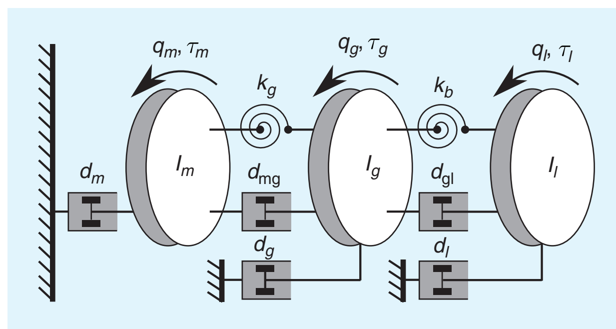


Figure 3. The linear mechanical system at the core of the Compliant Joint Toolbox.

The Compliant Joint Toolbox implements the linear mechanical dynamics in state-space form. The joint model has, in total, two inputs and generally seven outputs. The two inputs are the motor current and a disturbance input, which is either an externally applied load torque τ_l or load motion $\dot{q}_l$, depending on whether a locked-output model is chosen. The first three elements of the output vector are the three angles q_m, q_g, and q_l, and elements four to six are their derivatives. For convenience, the seventh output is the joint output torque applied to the load, following from

$$\tau_l = (q_g - q_l)k_b + (\dot{q}_g - \dot{q}_l)d_{gl}.$$

The benefit of the toolbox here is that the user can rapidly switch between mechanical and electrical models or compare actuator model structures against each other for identical parameter sets and within identical control schemes without manipulating equations or commenting/uncommenting duplicating source code.

Nonlinear Dynamics

Figure 4 illustrates the mechanical subsystem model realization with state vector x_q, system matrix $\mathbf{A}_q$, input and output matrices $\mathbf{B}_q$ and $\mathbf{C}_q$, and direct feedthrough matrix $\mathbf{F}_q$. Note, the symbol $\mathbf{F}_q$ for the feedthrough matrix deviates from the common usage in the control literature to better distinguish it from the damping matrix $\mathbf{D}$. Furthermore, for continuous-time models, $\mathbf{F}_q \equiv 0$. The input and output of this model are denoted by $\mathbf{u}_q$ and $\mathbf{y}_q$, respectively. The additive nonlinear dynamics term $\mathbf{g}(\mathrm{x}_q)$ in Figure 4 augments the linear state-space model; together, they represent the

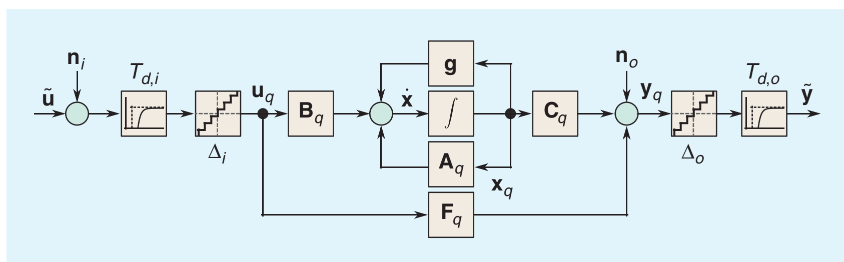


Figure 4. The model structure for the mechanical subsystem.

Algorithm 1: An Example of a Parameter File

```
% Save these lines in example_
params.m
% for later use in other examples.
%% Motor parameters
% Motor rotor inertia [kg m^2]
params.('I_m') = 9.407000e-02;
% Motor Damping [Nms/rad]
params.('d_m') = 2.188643e-03;
% Motor Coulomb damping [Nm]
params.('d_cm') = 2.6400;
% Torque constant [Nm/A]
params.('k_t') = 4.100000e-02;

%% Gear parameters
% Gear transmission ratio [.]
params.('n') = 100;
% Gearbox damping [Nms/rad]
params.('d_g') = 2.2000;
% Gearbox Damping - negative direc-
tion [Nms/rad]
params.('d_g_n') = 2.1000;
% Gearbox Coulomb damping [Nm]
params.('d_cg') = 3.2800;
%% Sensor parameters
% Sensor inertia [kg m^2]
params.('I_l') = 1.137000e-04;
% Sensor stiffness [Nm/rad]
params.('k_b') = 21000;
```

Algorithm 2: The Documentation Page of the `genericJoint` Class

```
% genericJoint class documentation
doc genericJoint
```

nominal system behavior. The following nonlinear effects are supported by the toolbox.

Asymmetric Viscous Friction

The most dominant nonlinear dynamics effect in torque-controlled actuators is friction. The parameters d_m, d_g, and d_l describe the symmetric linear viscous friction behavior in the support and transmission mechanisms. However, viscous friction may be modeled to be asymmetric with respect to the sign of the velocity.

(Asymmetric) Coulomb Friction

In addition to viscous friction, constant Coulomb friction is a nonlinear effect that dominates, especially the lower-speed regime of torque-controlled actuators, and it can also be asymmetric.

Torque Ripple

Apart from friction, torque ripples perturb the actuator torque generation. Multiple sources contribute to this effect, including commutation ripple, mutual torque ripple, cogging torque ripple, current offset ripple, gearbox teeth-meshing ripple, assembly eccentricity, and encoder ripple. All ripple sources combine to produce a ripple torque τ_r that is periodic with the rotor angle q_m. The Compliant Joint Toolbox incorporates ripple through a Fourier series in the rotor angle q_m. This ripple model is linear in the amplitude parameters A_j and B_j and considers a number N_ω of spatial ripple frequencies ω_q. As all nonlinear dynamics terms result in torque, they can be introduced into the models as an additional summand through $g(x_q)$ in the state equation, as indicated by Figure 4.

Noise, Quantization, and Delays

A use case of the toolbox is to simulate the nominal joint behavior to conceptually test and analyze controllers under ideal conditions. In nonideal cases, the actual system input and output are each subject to additive noise. The communication interfaces with the hardware introduce delays in the commands and measurements. Finite numeric data-type precision and converter and pulse-width modulation resolution introduce quantization. The Compliant Joint Toolbox allows the investigation of their impact on control performance as well as quick comparison with the ideal case.

Model Parameters

The starting point for modeling an actuator is a parameter file. Parameter files are nothing but m-scripts defining a struct named `param`. The class `genericJoint` assigns default values to all parameters; the parameter script is required only to specify deviations from these default values. Algorithm 1 shows an example of such a parameter file.

A full list and description of model parameters can be found on the documentation page of the `genericJoint` class (see Algorithm 2).

We save these parameters in an m-file, `example_params.m`. Moreover, the toolbox comes with a collection of detailed parameter file examples. Historically, they comprise parameters for TREE robotics actuators (https://www.treerobotics.eu). The files are located in the `param` subdirectory of the toolbox.

Model Generation

After collecting a joint's parameters in the `param` struct, the next step is to instantiate a `jointBuilder` that enables the generation of ready-to-use model classes. Once instantiated, the joint builder generates the model classes through the `buildJoint` method. The technique requires the parameter file and the desired linear dynamics to be specified. Here, we reuse the parameter file `example_params.m` created in the example in the "Model Parameters" section. Optionally, a cell list of nonlinear effects can be provided, and a custom class name (here, `Example_Joint`) can be specified (see Algorithm 3).

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 3: An Instantiated Joint Builder

```
%% Instantiate a jointBuilder
jb = jointBuilder;

%% Build joint model classes
% Here we reuse the parameters stored in
% example_params.m during the previous
example.
jb.buildJoint (...
'example_params',...      % parameters
'output_fixed_            % linear dynamics
rigid_gearbox',...
{'coulomb', 'vis-         % nonlinear dynamics
cous_asym'},...
'electric_dyn',...        % electro-dynamics
'Example_Joint');         % custom class name
```

Algorithm 4: The Method Documentation

```
%% buildJoint method documentation
doc jointBuilder/buildJoint
```

Possible values and combinations of the input parameters are detailed in the method documentation (see Algorithm 4).

After model generation, the `jointBuilder` build directory must be added to the MATLAB search path, and the freshly generated model class object can be instantiated. In the previous example, the generated model class was named `Example_Joint`, which can be instantiated as shown in Algorithm 5.

Algorithm 5: Adding the Joint Builder Directory to the MATLAB Search Path

```
%% Instantiate joint models
% add build directory to search path
addpath(jb.buildDir);
% create joint object
exJoint = Example_Joint;
```

Algorithm 6: Obtaining the Transfer Functions and State-Space Models

```
%% Transfer Functions of the Example
Joint
% Get all transfer functions
exTF = exJoint.getTF();
% Look at the torque output (row
index 7)
% w. r. to the input current (col
index 1)
exTF(7, 1)

% We obtain the same in the discrete
% time domain with:
exTFd = exJoint.getTFd();
exTFd(7, 1)

%% Get state-space system of Example
Joint
% Continuous-time
exSS = exJoint.getStateSpace();
% Discrete-time
exSSd = exJoint.getStateSpaceD();
```

Joint Model Analysis

This section demonstrates the use of the Compliant Joint Toolbox for the analysis of actuator models. Because of space constraints, a preview of the expected output is not shown here, but it can be found in the online documentation [23].

Linear Analysis

The `genericJoint` class builds upon the MATLAB core capabilities for numeric linear system analysis via transfer functions and state-space systems in continuous and discrete time. A benefit offered by the toolbox is that it removes the need to manually equate and insert the model parameters into the corresponding built-in MATLAB functions (`tf`, `ss`, and so forth). Using the generated classes, it offers direct access to the transfer functions and state-space matrices in the continuous- and discrete-time domains through a single line of code, independent of the selected model. This enables rapid switching and comparison of transfer functions or state-space matrices for different models and parameter sets. In linear analysis, nonlinear dynamics are linearized or ignored. (Coulomb friction is inherently not linearizable and is thus ignored; asymmetric viscous friction is made symmetric, and torque ripple is ignored.) The case in Algorithm 6 uses the previously generated joint class example and demonstrates how to obtain the transfer functions and state-space models.

Symbolic Equations

With the Symbolic Math Toolbox installed, the Compliant Joint Toolbox also allows inspection of the dynamics in symbolic form. This eases the analytical comprehension of how individual parameters affect the dynamics. In terms of implementation, the toolbox offers the `genericJoint` methods `makeSym` and `makeNum` to convert instances of joint models between numeric and symbolic representations. The case in Algorithm 7 considers the transfer function of the previous example, but this time in symbolic form.

Analysis Plots and Datasheet Generation

The `datasheetGenerator` class is instantiated for a given joint class and implements a public method interface to draw

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 7: The Transfer Function in Symbolic Form

```
%% Convert joint to symbolic form
exJoint.makeSym();

% Get all transfer functions
exTF = exJoint.getTF();

% Look at the input current (col index 1)
% to torque output (row index 7).
% Pretty print the result:
pretty(exTF(7, 1))

%% Return object to numeric form
exJoint.makeNum();
```

Algorithm 8: Assembling the Analysis into a PDF Datasheet

```
%% Generate a datasheet for the actuator
% Instantiate a dataSheetGenerator for the example
dsg = dataSheetGenerator(exJoint);
% Invoke datasheet generation
fName = dsg.generateDataSheet(); % look at output
% Look at the result
open(fName)
```

analysis plots illustrating torque speed and efficiency diagrams, thermal characteristics, and the torque bandwidth maps introduced in [5]. Provided that a LaTeX installation is present on the user's computer, the class can assemble the analyses into a PDF datasheet file summarizing the properties of the considered actuator. Algorithm 8 describes this procedure, reusing the joint class example created in the previous cases.

The method `generateDataSheet` executes routines to create and save the individual plots and generates and compiles a LaTeX file into a PDF document, exemplified in Figure 5 [24].

Simulink Library

The Compliant Joint Toolbox provides a Simulink library, `cjt_library`, located in the toolbox's `lib` directory. The library comprises three sublibraries for models, observers, and controllers. All blocks are Simulink Real-Time compatible, so they are suited for deployment on real, physical target hardware systems. The library blocks make use of the joint model classes generated by the joint builder. Their principal mask parameter is the user-specified joint object or joint class name. The blocks adapt their internal structure and behavior according to the dynamics and parameters specified in these derived joint classes. The following subsections detail each of the three sublibraries and report examples of their use in connection with real-world experimental setups.

Joint Model Blocks

The joint model library contains three blocks—two for the electrical subsystem and one for the mechanical subsystem. A signal bus `jointBus` serves as a common data structure to share joint-state information among the blocks. The latter may, however, be used independently or in conjunction with user-defined blocks.

The Electrical Subsystem Blocks

These allow the user to include input delays, quantization, and measurement noise to account for nonideal system behavior or, if desired, to simulate ideal system dynamics. The basic block models the single-phase electrical dynamics, providing a single-phase armature voltage input in addition to the joint bus. The more advanced version models the two-phase d–q plane electrical system and is suitable for vector control. For both, the block outputs are the winding currents and generated electromotive torque.

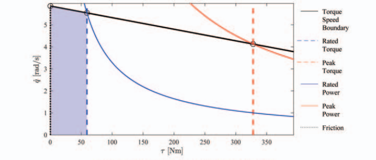
The Mechanical Subsystem Block

This block features a mask interface similar to the electrical subsystem blocks described previously, with a joint object or class name as the principal parameter. The user can enable input and/or output delays, noise, and quantization and can specify filter cutoff frequencies to realistically simulate velocity and torque readings from numerical differentiation. The mechanical subsystem is driven by the electrically generated torque. Depending on the chosen joint model structure, the second model input is a load torque or motion. The model output `jointBus` contains the joint states and output torque. When used in combination with an electrical subsystem block, the bus is fed back to the corresponding input of the electrical subsystem block so that the back electromotive force from the motion can be computed correctly.

Observers

In practice, the measurement of the entire actuator state is not always possible. For reasons of complexity and spatial, energy, and financial economy, developers typically seek to minimize the number of sensors used in an actuator. Dynamic effects, such as friction, external loads, and sensor imperfections, are difficult to model reliably and accurately. Redundancy and fault detection and isolation are crucial objectives in safety-critical robot operation, especially when operation occurs in the vicinity of humans. These aspects have led to the rigorous application of state and disturbance observers in compliant actuator control.

The Compliant Joint Toolbox features a Simulink blockset with four different observer implementations frequently found in the literature: the Luenberger observer, Kalman filter, generalized momentum disturbance observer [6], and linear transfer function disturbance observer [2], [7]. The inputs to all these blocks are the motor torque reference and



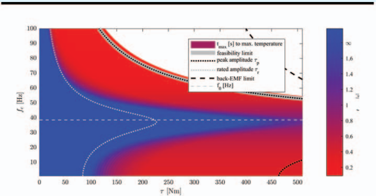


Figure 5. A datasheet example, automatically generated for the `Example_Joint` class using the `datasheetGenerator` class. A PDF with this datasheet is available online [24].

the `jointBus`. As output, they provide either a disturbance estimate or a state and output estimate. These four blocks are the core components of some of the controllers outlined in the next section.

Controllers

The controllers in the Compliant Joint Toolbox are implemented in discrete time. We provide an overview here. The simplest one provided is a pure desired-torque feedforward command that can be combined with an integral controller, as reported in [8]. As an alternative to integral action, [9] applied a linear disturbance observer, with the nominal plant performing only as a rotating mass to compensate for disturbances like friction.

The most common torque controller class in the literature is the proportional derivative (PD) type. For example, a pure PD torque controller was cascaded with an outer-loop PD position controller in [10]. A controller discussed in [8] augmented the PD feedback loop with a desired torque feedforward action. The controller presented in [11] supplemented the PD loop with a disturbance observer based on nominal open-loop plant dynamics. In contrast to [9], the authors of [11] incorporated linear viscous friction in the nominal plant model of the disturbance observer and added a feedforward nonlinear friction-compensation action. The authors of [3] proposed a disturbance observer based on a model of the nominal closed control loop, which augmented the PD torque controller. A disturbance

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Table 1. The linear mechanical subsystem models.

Load-Inertia Models

Full Dynamics

$$\mathbf{I}: \begin{bmatrix} I_m & 0 & 0 \\ 0 & I_g & 0 \\ 0 & 0 & I_l \end{bmatrix} \qquad \mathbf{D}: \begin{bmatrix} d_m + d_{mg} & -d_{mg} & 0 \\ -d_{mg} & d_g + d_{mg} + d_{gl} & -d_{gl} \\ 0 & -d_{gl} & d_l + d_{gl} \end{bmatrix}$$

$$\mathbf{q}: [q_m \;\; q_g \;\; q_l]^{\mathrm{T}} \qquad \mathbf{K}: \begin{bmatrix} k_g & -k_g & 0 \\ -k_g & k_g + k_b & -k_b \\ 0 & -k_b & k_b \end{bmatrix}$$

$$\mathbf{u}_q: [\tau_m \;\; \tau_l]^{\mathrm{T}} \qquad \mathbf{B}_q: \begin{bmatrix} k_g & -k_g & 0 \\ -k_g & k_g + k_b & -k_b \\ 0 & -k_b & k_b \end{bmatrix}$$

$$\mathbf{A}_q: \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -\frac{k_g}{I_m} & \frac{k_g}{I_m} & 0 & -\frac{d_m + d_{mg}}{I_m} & \frac{d_{mg}}{I_m} & 0 \\ \frac{k_g}{I_g} & -\frac{k_b + k_g}{I_g} & \frac{k_b}{I_g} & \frac{d_{mg}}{I_g} & -\frac{d_{mg} + d_g + d_{gl}}{I_g} & \frac{d_{gl}}{I_g} \\ 0 & \frac{k_b}{I_l} & -\frac{k_b}{I_l} & 0 & \frac{d_{gl}}{I_l} & -\frac{d_l + d_{gl}}{I_l} \end{bmatrix}$$

Rigid Gearbox $q_m \equiv q_g$

$$\mathbf{I}: \begin{bmatrix} I_m + I_g & 0 \\ 0 & I_l \end{bmatrix} \qquad \mathbf{D}: \begin{bmatrix} d_m + d_g + d_{gl} & -d_{gl} \\ -d_{gl} & d_l + d_{gl} \end{bmatrix}$$

$$\mathbf{q}: [q_m \;\; q_l]^{\mathrm{T}} \qquad \mathbf{K}: \begin{bmatrix} k_b & -k_b \\ -k_b & k_b \end{bmatrix}$$

$$\mathbf{u}_q: [\tau_m \;\; \tau_l]^{\mathrm{T}} \qquad \mathbf{B}_q: \begin{bmatrix} 0 & 0 & \frac{1}{I_g + I_m} & 0 \\ 0 & 0 & 0 & \frac{1}{I_l} \end{bmatrix}^{\mathrm{T}}$$

$$\mathbf{A}_q: \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{k_b}{I_g + I_m} & \frac{k_b}{I_g + I_m} & -\frac{d_m + d_g + d_{gl}}{I_g + I_m} & \frac{d_{gl}}{I_g + I_m} \\ \frac{k_b}{I_l} & -\frac{k_b}{I_l} & \frac{d_{gl}}{I_l} & -\frac{d_l + d_{gl}}{I_l} \end{bmatrix}$$

Fixed-Output Models

Rigid $q_m \equiv q_g \equiv q_l$

$$\mathbf{I}: I_\Sigma^{*} \qquad \mathbf{D}: d_\Sigma^{**}$$

$$\mathbf{q}: q_m \qquad \mathbf{K}: \infty$$

$$\mathbf{u}_q: \begin{bmatrix} \tau_m \\ \tau_l \end{bmatrix} \qquad \mathbf{B}_q: \begin{bmatrix} 0, & 0 \\ \frac{1}{I_\Sigma^{*}}, & \frac{1}{I_\Sigma^{*}} \end{bmatrix}$$

$$\mathbf{A}_q: \begin{bmatrix} 0 & 1 \\ 0 & -\frac{d_\Sigma^{*}}{I_\Sigma^{**}} \end{bmatrix}$$

Fixed-Output Full Dynamics

$$\mathbf{I}: \begin{bmatrix} I_m & 0 \\ 0 & I_g \end{bmatrix} \qquad \mathbf{D}: \begin{bmatrix} d_m + d_{mg} & -d_{mg} \\ -d_{mg} & d_g + d_{mg} + d_{gl} \end{bmatrix}$$

$$\mathbf{q}: [q_m \;\; q_l]^{T} \qquad \mathbf{K}: \begin{bmatrix} k_g & -k_g \\ -k_g & k_g + k_b \end{bmatrix}$$

$$\mathbf{u}_q: \begin{bmatrix} \tau_m \\ \dot{q}_l \end{bmatrix} \qquad \mathbf{B}_q: \begin{bmatrix} 0, & 0, & 0, & \frac{1}{I_m}, & 0 \\ 0, & 0, & 1, & 0, & \frac{d_{gl}}{I_g} \end{bmatrix}^{\mathrm{T}}$$

$$\mathbf{A}_q: \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -\frac{k_g}{I_m} & \frac{k_g}{I_m} & 0 & -\frac{d_m + d_{mg}}{I_m} & \frac{d_{mg}}{I_m} \\ \frac{k_g}{I_g} & -\frac{k_g + k_b}{I_g} & \frac{k_b}{I_g} & \frac{d_{mg}}{I_g} & -\frac{d_g + d_{mg} + d_{gl}}{I_g} \end{bmatrix}$$

Rigid Gearbox $q_m \equiv q_g$

$$\mathbf{I}: I_m + I_g \qquad \mathbf{D}: d_m + d_g + d_{gl}$$

$$\mathbf{q}: [q_m \;\; q_l]^{\mathrm{T}} \qquad \mathbf{K}: k_b$$

$$\mathbf{u}_q: \begin{bmatrix} \tau_m \\ \dot{q}_l \end{bmatrix} \qquad \mathbf{B}_q: \begin{bmatrix} 0, & 0, & \frac{1}{I_m + I_g} \\ 0, & 1, & \frac{d_{gl}}{I_m + I_g} \end{bmatrix}^{T}$$

$$\mathbf{A}_q: \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -\frac{k_b}{I_m + I_g} & \frac{k_b}{I_m + I_g} & -\frac{d_m + d_g + d_{gl}}{I_m + I_g} \end{bmatrix}$$

* $I_\Sigma = I_m + I_g + I_l$; ** $d_\Sigma = d_m + d_g + d_l$.

observer based on the closed control loop was adopted by [4] and [7] in the context of control for so-called reaction force series elastic actuators. The controller implemented this scheme based on a proportional–integral–derivative (PID) torque control loop with desired torque feedforward action. The controller in [12] also was a PID controller with a desired torque feedforward command, using the open-loop nominal plant model of a full-mass spring-damper system.

If full state information is available through measurement or reliable state observation, state feedback controllers, such as linear–quadratic regulator controllers, can be designed. The state feedback controller originally proposed in [13] was reformulated in a more general passivity-based torque and impedance control framework in [14]. During this process, the controller gains were redefined to yield a clear physical interpretation. In the context of the aforementioned controllers, the torque control part presented in [14] can be seen as a PD-type controller with positive direct torque feedback similar to [15]. The controller was augmented by a generalized momentum-based disturbance observer [6].

The blocks provided in the controller library implement, in discrete time, all the controllers just discussed. Controllers with an inner velocity- and/or position-control loop, such as reported in [16], have not been implemented so far, but there is no technical barrier to doing so. A template block serves as a starting point for users to develop new controllers.

Interfacing With Hardware

The Compliant Joint Toolbox was developed within the scope of [2], [5], and [17] and implemented on the WALK-MAN and TREE actuators. While it was initially developed for modeling and simulation, its rapid interfacing with real actuator hardware was truly helpful for data recording, testing, debugging, and tuning of joint torque controllers. The toolbox shrank the time and effort required to move from simulation to experiments. This became particularly useful when coping with different sizes and prototype stages of the actuators depicted in Figure 6. All of these actuators feature an industrial EtherCAT interface. However, from the toolbox side, there is no requirement to use EtherCAT; the toolbox can be used with whatever interface is supported by the Mathworks Simulink Real-Time application.

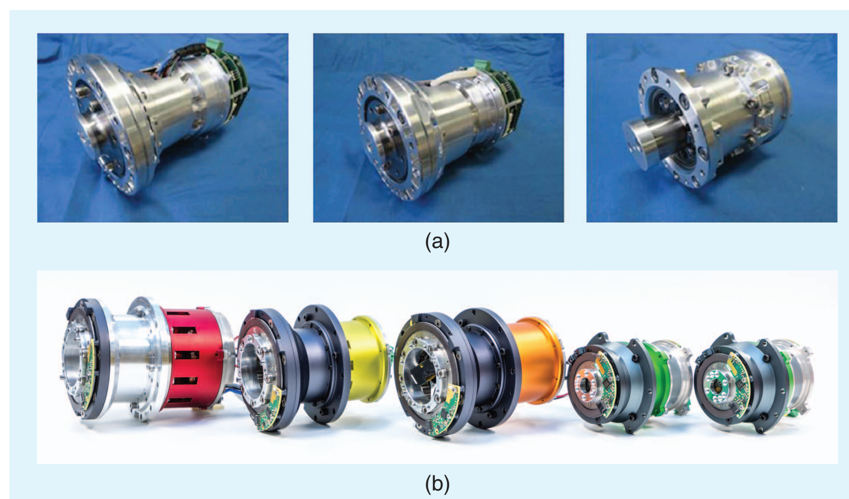


Figure 6. (a) The WALK-MAN actuators and (b) the TREE actuators with which the authors interfaced the Compliant Joint Toolbox.

Figure 7(a) presents an example of how to set up EtherCAT communication among the actuator, Simulink Real-Time target, and user console for a single actuator. A more detailed tutorial on how to organize an EtherCAT network is presented in [18]. The basic scheme—to create a Simulink block diagram that controls the actuator—is shown in Figure 7(b). It uses a controller block (blue) from the Compliant Joint Toolbox and the communication interface blocks (gray) provided by the Simulink Real-Time Toolbox.

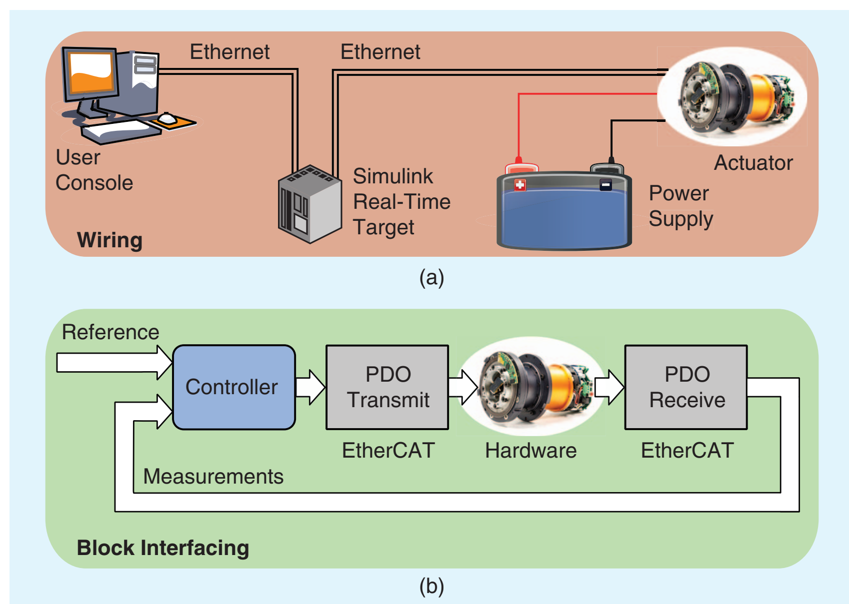


Figure 7. Examples for interfacing with actuator hardware on (a) the physical level and (b) the Simulink level. PDO: process data objects.

Summary and Future Directions

This article presented the Compliant Joint Toolbox and introduced its main concepts. The basic use of the toolbox was demonstrated, and code examples and references to more-detailed information were given.

We plan to extend the toolbox's capabilities to capture more nonlinear

dynamics effects, such as nonlinear stiffness curves, and more advanced friction models, including hysteresis with memory, as well as more usage examples. Furthermore, we aim to interface the Compliant Joint Toolbox with the Robotics Toolbox [19]. This will allow researchers to model and simulate full torque-controlled robotic systems and rapidly reach experimental readiness.

We hope the Compliant Joint Toolbox can catalyze the ongoing discussion on compliant robot actuation, support academic education in the field, and contribute to community efforts toward common notions, metrics, and benchmarks that ease torque-controlled actuation design, comparison, and selection across diverse robotic applications.

Our final words here are a call to action. The Compliant Joint Toolbox is open source, and we are happy to receive contributions from the community. Input is welcome in the form of code, feedback, discussion, bug reports, and feature requests.

Acknowledgments

We would like to thank Peter Corke of the Queensland University of Technology for his valuable and encouraging feedback. This work received financial support from European Research Council projects under the European Union's Seventh Framework Program grant 611832 (WALK-MAN) as well as under the Horizon 2020 research and innovation program, grants 644839 (CENTAURO) and 644727 (CogIMon).

References

[1] L. Meirovitch, *Fundamentals of Vibrations*. Boston: McGraw-Hill, 2001.

[2] W. Roozing, J. Malzahn, D. G. Caldwell, and N. G. Tsagarakis, "Comparison of open-loop and closed-loop disturbance observers for series elastic actuators," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2016, pp. 3842–3847.

[3] K. Kong, J. Bae, and M. Tomizuka, "Control of rotary series elastic actuator for ideal force-mode actuation in human-robot interaction applications," *IEEE/ASME Trans. Mechatronics*, vol. 14, no. 1, pp. 105–118, 2009.

[4] N. Paine et al., "Actuator control for the NASA-JSC Valkyrie humanoid robot," *J. Field Robot.*, vol. 32, no. 3, pp. 378–396, 2015.

[5] J. Malzahn, N. Kashiri, W. Roozing, N. Tsagarakis, and D. Caldwell, "What is the torque bandwidth of this actuator?" in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2017, pp. 4762–4768.

[6] L. Le Tien, A. Albu-Schäffer, A. De Luca, and G. Hirzinger, "Friction observer and compensation for control of robots with joint torque measurement," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2008, pp. 3789–3795.

[7] N. Paine, S. Oh, and L. Sentis, "Design and control considerations for high-performance series elastic actuators," *IEEE/ASME Trans. Mechatronics*, vol. 19, no. 3, pp. 1080–1091, 2014.

[8] D. Vischer and O. Khatib, "Design and development of high-performance torque-controlled joints," *IEEE Trans. Robot. Autom.* vol. 11, no. 4, pp. 537–544, 1995.

[9] K. Kaneko, S. Kondo, and K. Ohnishi, "A motion control of flexible joint based on velocity estimation," in *Proc. IEEE Annu. Conf. Industrial Electronics Society (IECON)*, 1990, pp. 279–284.

[10] M. C. Readman, *Flexible Joint Robots*. Boca Raton, FL: CRC, 1994.

[11] H. S. Lee and M. Tomizuka, "Robust motion controller design for high-accuracy positioning systems," *IEEE Trans. Ind. Electron.*, vol. 43, no. 1, pp. 48–55, 1996.

[12] M. A. Hopkins, S. A. Ressler, D. F. Lahr, A. Leonessa, and D. W. Hong, "Embedded joint-space control of a series elastic humanoid," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2015, pp. 3358–3365.

[13] A. Albu-Schäffer and G. Hirzinger, "State feedback controller for flexible joint robots: A globally stable approach implemented on DLR's light-weight robots," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, vol. 2, 2000, pp. 1087–1093.

[14] A. Albu-Schaffer, C. Ott, and G. Hirzinger, "A unified passivity-based control framework for position, torque and impedance control of flexible joint robots," *Int. J. Robot. Res.*, vol. 26, no. 1, pp. 23–39, 2007.

[15] M. Hashimoto, T. Horiuchi, Y. Kiyosawa, and H. Hirabayashi, "The effects of joint flexibility on robot motion control based on joint torque positive feedback," in *Proc. IEEE Int. Conf. Robotics and Automation*, 1991, pp. 1220–1225.

[16] G. Wyeth, "Control issues for velocity sourced series elastic actuators," in *Proc. Australasian Conf. Robotics and Automation*, Auckland, New Zealand, 2006.[Online]. Available: http://www.araa.asn.au/conferences/acra-2006/table-of-contents/

[17] W. Roozing, J. Malzahn, N. Kashiri, D. G. Caldwell, and N. G. Tsagarakis, "On the stiffness selection for torque controlled series-elastic actuators," *IEEE Robot. Autom. Lett.*, vol. 2, no. 4, pp. 2255–2262, 2017.

[18] K. Langlois et al., "EtherCAT tutorial: An introduction for real-time hardware communication on Windows [Tutorial]," *IEEE Robot. Autom. Mag.*, vol. 25, no. 1, pp. 22–122, 2018.

[19] P. I. Corke, *Robotics, Vision, and Control: Fundamental Algorithms in MATLAB*, 2nd ed. (Springer Tracts in Advanced Robotics, no. 118). Cham, Switzerland: Springer-Verlag, 2017.

[20] J. Malzahn and W. Roozing, "Code capsule to try the compliant joint toolbox," 2018. [Online]. Available: https://codeocean.com/2018/08/02/compliant-joint-toolbox-capsule/code

[21] J. Malzahn and W. Roozing, "Compliant Joint Toolbox—Getting started," Aug. 2, 2018. [Online]. Available: https://github.com/geez0x1/CompliantJointToolbox/wiki/Getting-Started

[22] J. Malzahn and W. Roozing, "Technical documentation," Nov. 22, 2018. [Online]. Available: https://github.com/geez0x1/CompliantJointToolbox/wiki/Technical-Documentation

[23] J. Malzahn and W. Roozing, "Compliant Joint Toolbox—Actuator analysis," Nov. 16, 2018. [Online]. Available: https://github.com/geez0x1/CompliantJointToolbox/wiki/Actuator-Analysis

[24] J. Malzahn and W. Roozing, "Compliant Joint Toolbox—Example datasheet," [Online]. Available: https://github.com/geez0x1/CompliantJointToolbox/files/2590377/Example_Joint_datasheet.pdf

Jörn Malzahn, Humanoids and Human-Centered Mechatronics Lab, Istituto Italiano di Tecnologia, Genoa, Italy. Email: jorn.malzahn@iit.it.

Wesley Roozing, Robotics and Mechatronics, University of Twente, The Netherlands. Email: w.roozing at utwente.nl.

Nikos Tsagarakis, Humanoids and Human-Centered Mechatronics Lab, Istituto Italiano di Tecnologia, Genoa, Italy. Email: nikos.tsagarakis@iit.it.