

The Compliant Joint Toolbox for MATLAB: An Introduction with Examples

Jörn Malzahn[†], Wesley Roozing[‡] and Nikos Tsagarakis[†]

Abstract—**This paper presents the *Compliant Joint Toolbox* for modelling, 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 of ready-to-use joint actuator model classes from multiple parameter sets, incorporating a variety of nonlinear dynamics effects. The *Compliant Joint 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.**

***Index Terms*—torque controlled actuation, compliant actuators, rapid control prototyping.**

I. Introduction

Actuators are the central components that make robots move. Novel applications for robot technology demand fundamentally different actuation design paradigms and techniques from 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 every-day households. Actuators, apart from being the "movers" as with conventional bulky position controlled industrial robots, now become central components that also make robots "feel" interaction forces.

Recent developments in 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, development of actuators for robotic devices relies largely on the intuition and experience of an engineer rather than on any rigorous theory that guides proper balancing of the aforementioned demands along with also other criteria such as peak power and maximum load capacity as well as torque and motion bandwidth or impact resilience. Literature lacks proper understanding of relevant requirements along with metrics for their quantification to guide this process. Conventional notions (power-density, peak torque, maximum speed, single numbers for bandwidths etc.) are insufficient for new applications that are dominated by physical interaction.

The *Compliant Joint Toolbox* is available on GIT under the GNU General Public License v3.0. The toolbox can also be inspected on Code Ocean[1]. It has emerged during the authors' work on actuator modelling, design and control towards solutions for the previously discussed actuator design challenges in diverse robotic applications. While the first use was modelling 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 has helped the authors cope with a variety of actuator configurations in terms of e.g. rotor, gearbox, and torque sensor combinations, during the development of the WALK-MAN[2], CENTAURO[3] and CogIMon[4] robots. 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 has been an essential tool in the preparation of publications on modelling, 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, which is a proprietary software suite for technical computing and rapid algorithm prototyping. The MATLAB language is interpreted, requires low learning efforts, ships with numerous state-of-the-art algorithms and visualization tools and therefore has a short time to productivity.

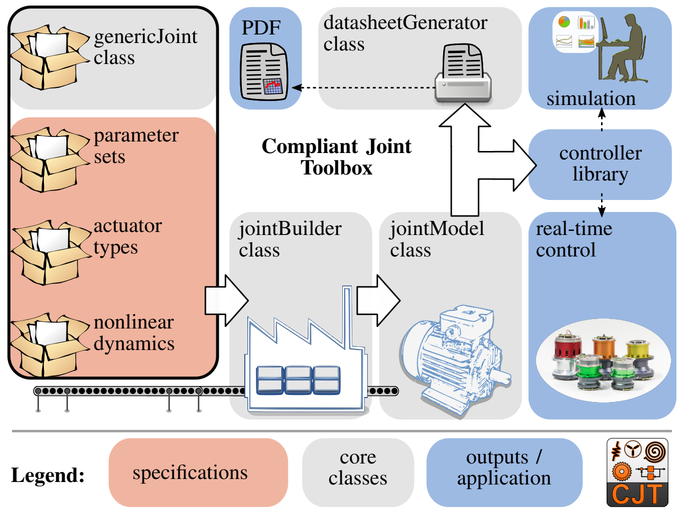


Fig. 1. The *Compliant Joint Toolbox* architecture adopting a variant of the abstract "factory creational design" pattern.

[†]Humanoids and Human Centred Mechatronics Lab, Istituto Italiano di Tecnologia (IIT), Via Morego 30, 16163 Genoa, Italy. E-mail: {jorn.malzahn,nikos.tsagarakis}@iit.it.

[‡]Robotics & Mechatronics (RaM), University of Twente, The Netherlands. E-mail: w.roozing at utwente.nl.

Manuscript received April 19, 2005; revised August 26, 2015.

[1]https://codeocean.com/2018/08/02/compliant-joint-toolbox-capsule/code

[2]www.walk-man.eu

[3]www.centauro-project.eu

[4]https://cogimon.eu/

The Python language shares most of the features above. Being non-proprietary, Python would have been the authors' preferred choice to implement *Compliant Joint Toolbox* and make it available to the community entirely for free. However, the crucial aspect that triggered the decision against a purely non-proprietary solution is the lack of a mature and sufficiently powerful open alternative for the features offered by the Simulink Real-Time Toolbox. It offers the chance to quickly interface with the actuator hardware based on standard industrial protocols such as CAN, Ethernet, and EtherCAT. This allows to rapidly develop, deploy, tune and test models and controllers on different actuator hardware and even with the actuator hardware-in-the-loop. Minimizing time and effort required to port developed concepts from simulation to experiments improves realism in research.

The authors hope the *Compliant Joint Toolbox* can catalyse the ongoing discussion on compliant robot actuation, support academic education in the field and contribute to community efforts towards common notions, metrics, and benchmarks that ease torque controlled actuation design, comparison and selection across diverse robotic applications. Hence, the motivation to make *Compliant Joint Toolbox* public and to draw community attention on it.

The next Section provides an overview of the *Compliant Joint Toolbox* architecture and core modules. Sec. III exemplifies the automated generation of ready-to-use joint model classes from actuator parameter sets. Sec. IV introduces the tools provided to analyse and compare joint model and controller performance. The simulation of models and controllers for single joints and complete robotic systems are demonstrated in Sec. V. Finally, Sec. VI exemplifies the toolbox use to interface with hardware. The paper concludes with a summary and perspectives for future developments in Sec. VII. All examples are designed and formatted to be copy-pasted from the paper to facilitate experimenting with the *Compliant Joint Toolbox* while reading this paper.

II. An Overview – The Toolbox Architecture

The toolbox architecture is illustrated in Fig. 1. 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 CJT notion of an actuator model comprises a mathematical structure (i.e. mathematical formulas) along with a set of values for the parameters present in the mathematical structure. As a consequence, two models with different mathematical structure can have the same values for their mutually common parameters. Vice versa, 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 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`, serves to automate datasheet compilation for the implemented joint models, providing an immediate picture of the joint's capabilities using 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, how to obtain and set up the toolbox.

A. 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 Sec. III-B.

The methods of `genericJoint` offer the conversion between discrete and continuous time representations of state space models and transfer functions, transform reflected inertia and friction parameters between motor and load side, compute motor characteristics such as torque-speed slope, no-load speed, stall torque, etc., and convert between numeric and symbolic model representations[5].

B. The `jointBuilder` Class

The number and complexity of the relevant dynamic effects vary from actuator to actuator and depending on the user's design or control objective. Furthermore, the same principal dynamics lead to diverse results when parametrized differently. As indicated in the bottom center of Fig. 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, *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. The `jointBuilder` usage is exemplified in Sec. III-C.

C. The `datasheetGenerator` Class

The `datasheetGenerator` generates 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 comprise a table listing the parameters of the actuator along with detailed descriptions for each individual parameter. Figures display the actuator characteristics such as the torque-speed curve, efficiency curve, torque bandwidth, thermal operation characteristics, etc. These plots are detailed in Sec. IV. The basic usage of `datasheetGenerator` is demonstrated in Sec. IV-C and an example datasheet is provided. Apart from the generation of fully formatted datasheets, visualization functionality to produce the embedded graphs for custom analysis is available to the user through a public method interface.

[5] Symbolic Math Toolbox™ required.

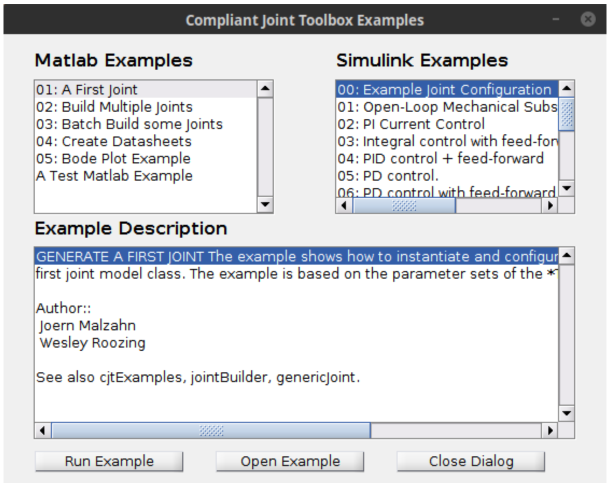


Fig. 2. The example browser (`cjtExamples.m`) allows to browse through available MATLAB and Simulink examples, and inspect or directly run them.

D. Simulation & Control

Being a result of research efforts on modelling, 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 are detailed in Sec. V-C. The controllers are implemented in discrete time masked Simulink blocks and use previously generated joint model classes for configuration. This way the Simulink code generation and real-time control features[6] can be exploited with the *Compliant Joint Toolbox* to rapidly prototype a control system for an experimental hardware setup. An example is described in Sec. VI.

E. Getting Started

To get started, all that is necessary is to obtain the toolbox code from its Git repository:

```
https://github.com/geez0x1/
    CompliantJointToolbox
```

To add the *Compliant Joint Toolbox* to your MATLAB search path, change your current MATLAB working directory to the toolbox directory and run `setCJTPaths.m`. You can get started from the available examples by using the example browser displayed in Fig. 2 or by following the Quickstart guide, which is provided online[7] as a short version of the complete toolbox documentation[8]. Alternatively, the toolbox can also be inspected and run completely contained on Code Ocean[9].

III. 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 out of them. The following Sections provide a number code examples, which can be found in `Take_a_Tour.m`.

[6]Mathworks® Real-time and/or Coder Toolboxes required.

[7]https://github.com/geez0x1/CompliantJointToolbox/wiki/Getting-Started

[8]https://github.com/geez0x1/CompliantJointToolbox/wiki/Technical-Documentation

[9]https://codeocean.com/2018/08/02/compliant-joint-toolbox-capsule/code

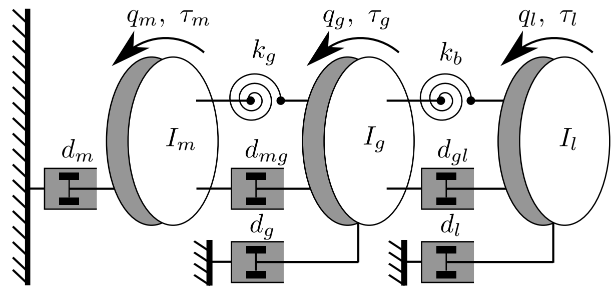


Fig. 3. Linear mechanical system at the core of the *Compliant Joint Toolbox*.

A. Generic Model Implementation

The linear electrical and linear mechanical subsystem models 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 behaviour, which allows to capture a broad variety of practically relevant nonlinear dynamic effects.

1) The Electrical Subsystem: The most common electrical drive in torque controlled robotic actuators are brushless DC motors (BLDC), 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 intended to specifically analyse current control performance or its implications on higher level controllers, the electrical dynamics can be neglected with respect to the mechanical time constant. This substantially shortens simulation time. Hence, by default a static model is used in built actuator models. Subsection III-C describes how to switch to a dynamic model.

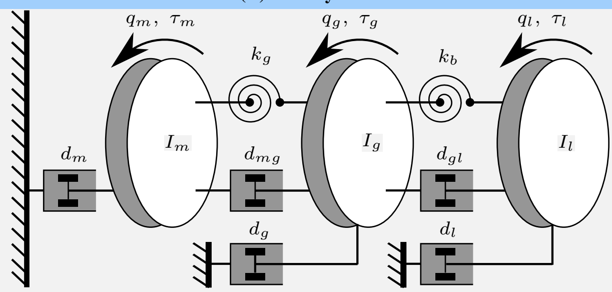
2) The Mechanical Subsystem: The mechanical subsystem is modelled as a chain of rotating masses interconnected via massless spring-damper elements as depicted in Fig. 3. The electrical drive rotor is an inertia I_m, which experiences a damping d_m with respect to ground. The gearbox contributes an inertia 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 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. The torques acting on the motor, gearbox and load are τ_m, τ_g and τ_l.

The linear equations of motion for this three-mass-system are straightforward to derive from first principles and can even be found in many textbooks on control or structural dynamics such as for example [1]. The *Compliant Joint Toolbox* features

TABLE I
LINEAR MECHANICAL SUBSYSTEM MODELS.

Load Inertia Models

(A) Full Dynamics



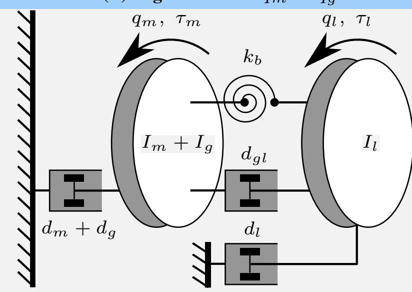
$$\mathbf{I}:\ \begin{bmatrix} I_m & 0 & 0 \\ 0 & I_g & 0 \\ 0 & 0 & I_l \end{bmatrix} \qquad \mathbf{D}:\ \begin{bmatrix} d_m + d_{mg} & -d_{mg} & 0 \\ -d_{mg} & d_g + d_{mg} + d_{gl} & -d_{gl} \\ 0 & -d_{gl} & d_l + d_{gl} \end{bmatrix}$$

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

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

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

(B) Rigid Gearbox $q_m \equiv q_g$



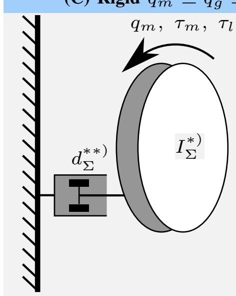
$$\mathbf{I}:\ \begin{bmatrix} I_m + I_g & 0 \\ 0 & I_l \end{bmatrix} \qquad \mathbf{D}:\ \begin{bmatrix} d_m + d_g + d_{gl} & -d_{gl} \\ -d_{gl} & d_l + d_{gl} \end{bmatrix}$$

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

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

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

(C) Rigid $q_m \equiv q_g \equiv q_l$



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

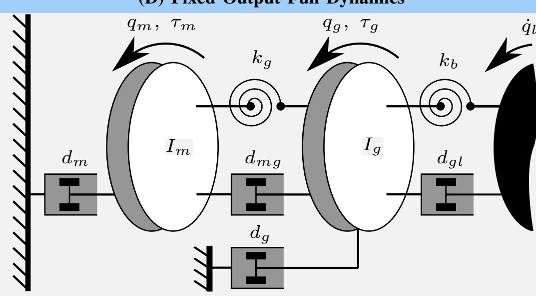
$$\mathbf{q}:\ q_m \qquad \mathbf{K}:\ \infty$$

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

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

Fixed Output Models

(D) Fixed Output Full Dynamics



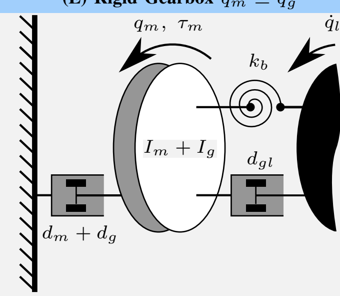
$$\mathbf{I}:\ \begin{bmatrix} I_m & 0 \\ 0 & I_g \end{bmatrix} \qquad \mathbf{D}:\ \begin{bmatrix} d_m + d_{mg} & -d_{mg} \\ -d_{mg} & d_g + d_{mg} + d_{gl} \end{bmatrix}$$

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

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

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

(E) Rigid Gearbox $q_m \equiv q_g$



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

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

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

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

*) $I_\Sigma = I_m + I_g + I_l$, **) $d_\Sigma = d_m + d_g + d_l$

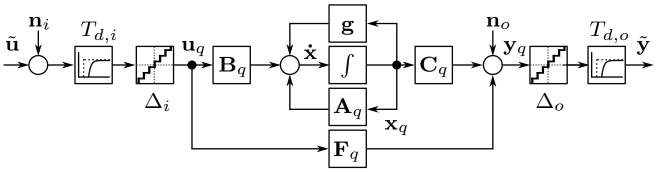


Fig. 4. Model structure for the mechanical subsystem.

several variants to this general model structure, such as a rigid gearbox, complete rigidity (single moving mass with friction), and fixed output configurations. In the latter, the load motion is defined by an external source, effectively allowing to connect the actuator model to 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].

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 the 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 and commenting/un-commenting duplicating source code.

3) Nonlinear Dynamics: Fig. 4 illustrates the mechanical subsystem model realization with state vector $\mathbf{x}_q$, system matrix $\mathbf{A}_q$, as well as input and output matrices $\mathbf{B}_q$, $\mathbf{C}_q$, and direct feed-through matrix $\mathbf{F}_q$[10]. The input and output of this model are denoted by $\mathbf{u}_q$ and $\mathbf{y}_q$, respectively. The additive nonlinear dynamics term $\mathbf{g}(\mathbf{x}_q)$ in Fig. 4 augments the linear state space model and together they represent the nominal system behaviour. The following nonlinear effects are supported by the toolbox:

a) 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 symmetric linear viscous friction behaviour in the support and transmission mechanisms. However, viscous friction may be modelled to be asymmetric with respect to the sign of the velocity.

b) (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 can also be asymmetric.

[10]The symbol $\mathbf{F}_q$ for the feedthrough matrix deviates from from the usage in common control literature to better distinguish it from the damping matrix $\mathbf{D}$. Furthermore, for continuous-time models $\mathbf{F}_q \equiv 0$.

c) Torque Ripple: Apart from friction, *torque ripples* perturb the actuator torque generation. Multiple sources contribute to the effect. Commutation ripple, mutual torque ripple, cogging torque ripple, current offset ripple, gearbox teeth meshing ripple, assembly eccentricity, encoder ripple. All ripple sources accumulate to 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 $\mathbf{g}(\mathbf{x}_q)$ in the state equation, as indicated by Fig. 4.

4) Noise, Quantization and Delays: A use case of the toolbox is to simulate the nominal joint behaviour to conceptually test and analyse controllers under ideal conditions. In the non-ideal case, 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, converter- and PWM-resolution introduce quantization. The *Compliant Joint Toolbox* allows to investigate their impact on control performance and quick comparison to the ideal case.

B. Model Parameters

The starting point to model 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 only required to specify deviations from these default values. An example of such a parameter file is given in the following:

```
% 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 direction [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;
```

A full list and description of model parameters can be found on the documentation page of the `genericJoint` class:

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

We save these parameters in an m-file `example_params.m`. Moreover, the toolbox comes with a collection of detailed example parameter files. Historically, they comprise parameters for TREE Robotics Actuators[11]. The files are located in the `param` subdirectory of the toolbox.

C. 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 method 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 Sec. III-B. Optionally, a cell list of nonlinear effects can be provided and a custom class name (here: `Example_Joint`) can be specified.

```
%% 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_rigid_gearbox' , ... % linear dynamics
{'coulomb', 'viscous_asym'}, ...    % nonlinear dynamics
'electric_dyn', ...                 % electro—dynamics
'Example_Joint');                   % custom class name
```

Possible values and combinations the input parameters are detailed in the method documentation:

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

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 follows:

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

IV. Joint Model Analysis

This section demonstrates the use of the *Compliant Joint Toolbox* for the analysis of actuator models. Due to space constraints, a preview of the expected output is not shown here, but can be found in the online documentation[12].

A. 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 to remove the need to manually equate and insert the model parameters into the corresponding built-in MATLAB functions (`tf`, `ss`, etc.). Using the generated classes it offers direct access to the transfer functions and state-space matrices in continuous and discrete time domain 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 linearised or ignored[13]. The example below uses the earlier generated example joint class and demonstrates how to obtain 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();
```

B. Symbolic Equations

With the Symbolic Math Toolbox™ installed, the *Compliant Joint Toolbox* allows inspection of the dynamics also in symbolic form. This eases the analytical understanding of how individual parameters affect the dynamics. Implementation wise, the toolbox offers the `genericJoint` methods `makeSym` and `makeNum`, to convert instances of joint models between numeric and symbolic representations. The following example considers the transfer function of the previous example, but this time 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();
```

C. Analysis Plots & Datasheet Generation

The `datasheetGenerator` class is instantiated for a given joint class and implements a public method interface to draw analysis plots illustrating torque-speed and efficiency diagrams, thermal characteristics, as well as the torque-bandwidth

[11] https://www.treerobotics.eu

[12] https://github.com/geez0x1/CompliantJointToolbox/wiki/Actuator-Analysis

[13] Coulomb friction is inherently not linearisable and thus ignored, asymmetric viscous friction is made symmetric, and torque ripple is ignored.

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. The following example describes this procedure, reusing the example joint class created in the previous examples:

```
%% 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)
```

The method `generateDataSheet` executes routines to create and save the individual plots, and generates and compiles a LaTeX file into a PDF document that is exemplified by Fig. 5.

V. Simulink Library

The *Compliant Joint Toolbox* provides a Simulink library named `cjt_library`, which is located in the `lib` directory of the toolbox. The library comprises three sub-libaries for models, observers, and controllers. All blocks are Simulink Real-Time compatible, so that 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 user-specified joint object or joint class name. The blocks adapt their internal structure and behaviour according to the dynamics and parameters specified in these derived joint classes. The following subsections detail each of the three sub-libraries and report examples of their usage in connection with real-world experimental setups.

A. 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 blocks. The blocks may however be used independently, or in conjunction with user-defined blocks.

1) The electrical subsystem blocks: allow the user to include input delays, quantization, and measurement noise, to account for non-ideal system behaviour or to simulate ideal system dynamics if desired. 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, suitable for vector control. For both, the block outputs are the winding current(s) and generated electromotive torque.

2) The mechanical subsystem block: features a similar mask interface to the electrical subsystem blocks described above, with a joint object or class name as principal parameter. The user can enable input and/or output delays as well as noise and quantization, and specify filter cut-off 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 (Table I), 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 back-EMF from motion can be computed correctly.

B. Observers

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

The *Compliant Joint Toolbox* features a Simulink blockset with four different observer implementations that are frequently found in literature: the Luenberger observer, Kalman filter, generalized momentum disturbance observer [6], and a linear transfer function disturbance observer [2], [7]. The inputs to all blocks are the motor torque reference and the `jointBus`. As output they provide either a disturbance estimate or state and output estimate. These four blocks are core components of some of the controllers outlined in the next subsection.

C. Controllers

The controllers contained in the *Compliant Joint Toolbox* are implemented in discrete time. We provide an overview here. The simplest controller 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] applies a linear disturbance observer with the nominal plant to be just a rotating mass to compensate for disturbances such as friction.

The most common torque controller class in literature is of PD type. For example, a pure PD torque controller is cascaded with an outer loop PD position controller in [10]. A controller discussed in [8] augments the PD feedback loop with a desired torque feedforward action. The controller presented in [11] supplements the PD loop with a disturbance observer based on nominal open loop plant dynamics. In contrast to [9], the authors of [11] incorporate the linear viscous friction in the nominal plant model of the disturbance observer, and add 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 augments the PD torque controller. A disturbance observer based on the closed control loop is adopted by [4], [7] in the context of the control for so-called reaction force series elastic actuators.

[15] https://github.com/geez0x1/CompliantJointToolbox/files/2590377/Example_Joint_datasheet.pdf

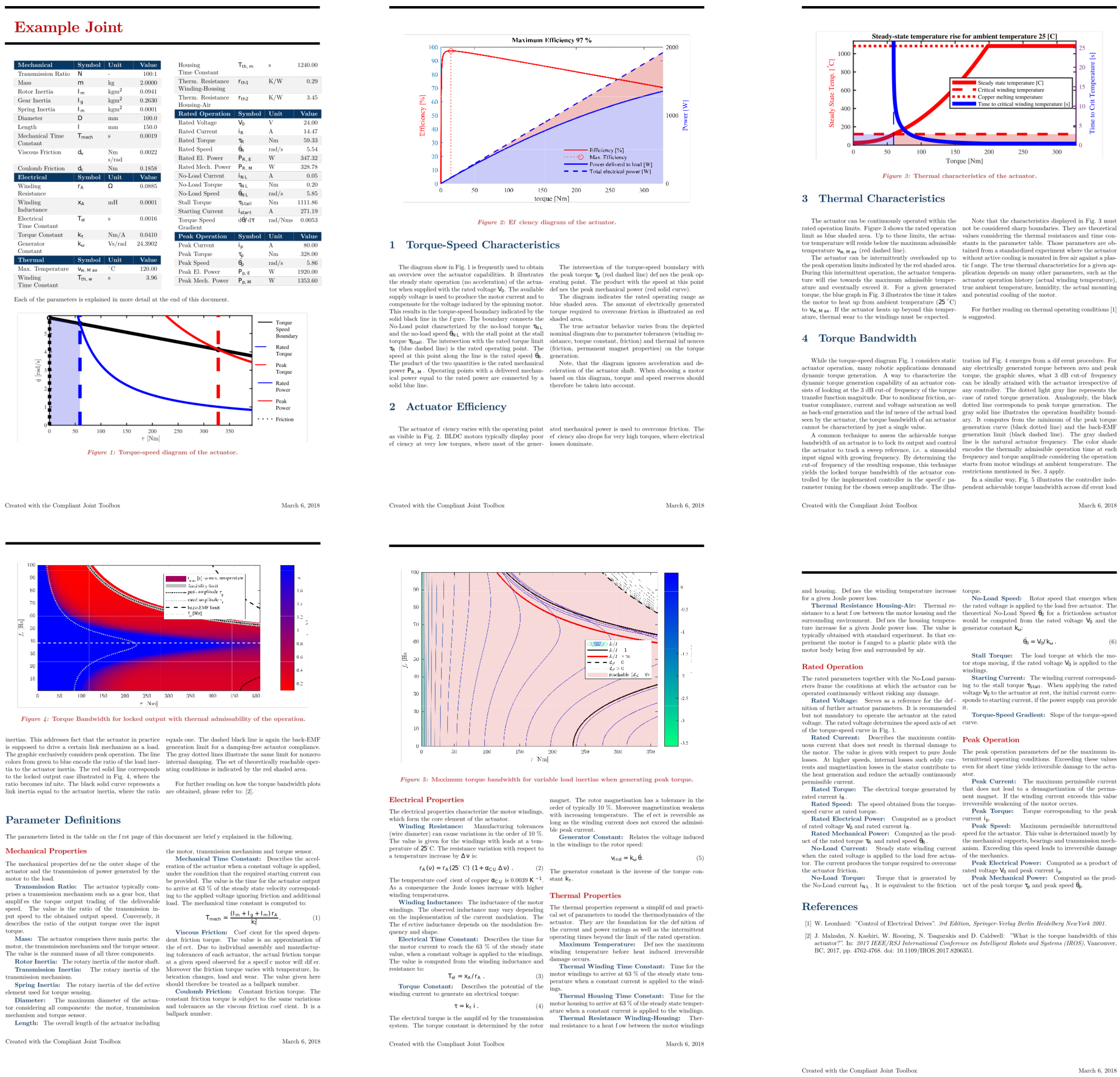


Fig. 5. Example datasheet, automatically generated for the `Example_Joint` class using the `datasheetGenerator` class. A PDF with this datasheet is available online[15]

The controller implements this scheme based on a PID torque control loop with desired torque feedforward action. The controller in [12] is a PID controller with desired torque feedforward command as well, and uses 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 LQR controllers can be designed. The state feedback controller originally proposed in [13] was reformulated within a more general passivity based torque and impedance control framework in [14]. During this process, the controller gains have been 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 has been augmented by a generalized momentum based disturbance observer [6].

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

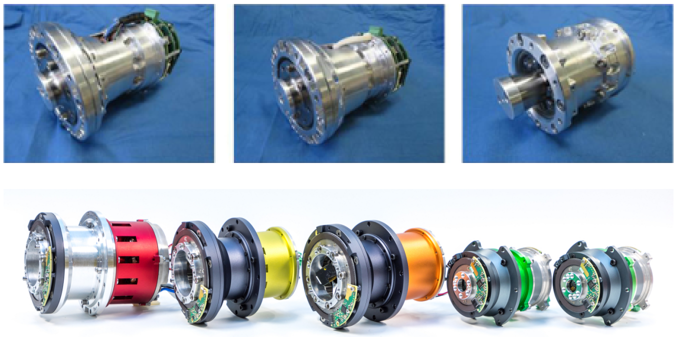


Fig. 6. The WALK-MAN (top) and TREE actuators (bottom) the authors interfaced the *Compliant Joint Toolbox* with.

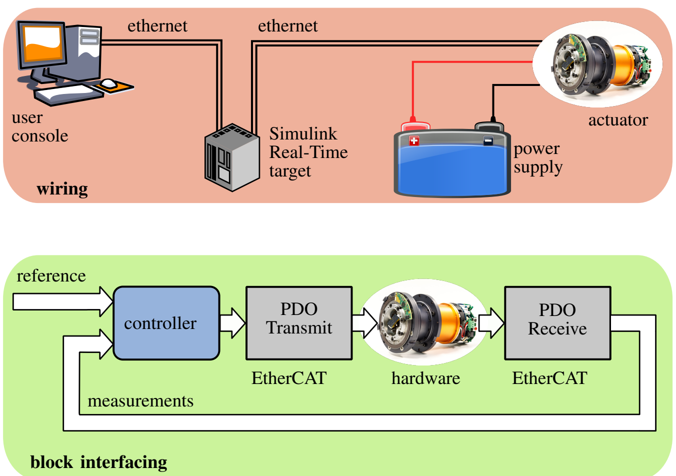


Fig. 7. Examples for interfacing with actuator hardware on the physical (top) and Simulink level (bottom).

VI. Interfacing with Hardware

The *Compliant Joint Toolbox* has been developed within the scope of the works [2], [5], [17] carried out on the WALK-MAN and TREE actuators. Initially developed for modelling and simulation, rapidly 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 Fig. 6. All 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 toolbox.

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

VII. Summary and Future Directions

This article presents the *Compliant Joint Toolbox*, and introduces the reader to its main concepts. The basic usage of the toolbox is demonstrated with code examples and references to more detailed information are given.

The authors plan to extend the toolbox capabilities to capture more nonlinear dynamics effects such as nonlinear stiffness curves and more advanced friction models including hysteresis with memory, together with more usage examples. Furthermore, they aim to interface the *Compliant Joint Toolbox* with the Robotics Toolbox [19]. This will allow to model, simulate, and rapidly reach experimental readiness of full torque controlled robotic systems.

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

The final words of this article are a call to action. The *Compliant Joint Toolbox* is open source and happy to receive contributions from the community. Contributions are welcome in the form of code, feedback, discussion as well as bug reports and feature requests.

Acknowledgment

The authors would like to thank Peter Corke from the Queensland University of Technology for his valuable and encouraging feedback.

This work has received financial support from European Research Council projects WALK-MAN (no. 611832), CENTAURO (no. 644839) and CogIMon (no. 644727).

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 *International Conference on Intelligent Robots and Systems*. IEEE/RSJ, 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 Transactions on Mechatronics*, vol. 14, no. 1, pp. 105–118, Feb. 2009.

[4] N. Paine *et al.*, "Actuator Control for the NASA-JSC Valkyrie Humanoid Robot," *Journal of Field Robotics*, vol. 32, no. 3, pp. 378–396, May 2015.

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

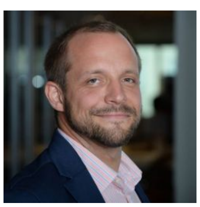
[6] L. Le Tien, A. Albu-Schäffer, A. De Luca, and G. Hirzinger, "Friction observer and compensation for control of robots with joint torque measurement," in *International Conference on Intelligent Robots and Systems IROS*. IEEE/RSJ, 2008, pp. 3789–3795.

[7] N. Paine, S. Oh, and L. Sentis, "Design and Control Considerations for High-Performance Series Elastic Actuators," *IEEE/ASME Transactions on Mechatronics*, vol. 19, no. 3, pp. 1080–1091, Jun. 2014.

[8] D. Vischer and O. Khatib, "Design and development of high-performance torque-controlled joints," *IEEE Transactions on Robotics and Automation*, 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 *Annual Conference of the Industrial Electronics Society (IECON)*. IEEE, 1990, pp. 279–284.

[10] M. C. Readman, *Flexible Joint Robots*. CRC press, 1994.
[11] H. S. Lee and M. Tomizuka, "Robust motion controller design for high-accuracy positioning systems," *IEEE Transactions on Industrial Electronics*, 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 *International Conference on Intelligent Robots and Systems (IROS)* . IEEE/RSJ, 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 lightweight robots," in *IEEE/RSJ International Conference on 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," *The International Journal of Robotics Research*, vol. 26, no. 1, pp. 23–39, Jan. 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 *International Conference on Robotics and Automation.* IEEE, 1991, pp. 1220–1225.
[16] G. Wyeth, "Control issues for velocity sourced series elastic actuators," in *Australasian Conference on Robotics and Automation*. Australian Robotics and Automation Association Inc, 2006.
[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 Robotics and Automation Letters*, 2017.
[18] K. Langlois, T. van der Hoeven, D. Rodriguez Cianca, T. Verstraten, T. Bacek, B. Convens, C. Rodriguez-Guerrero, V. Grosu, D. Lefeber, and B. Vanderborght, "EtherCAT Tutorial: An Introduction for Real-Time Hardware Communication on Windows [Tutorial]," *IEEE Robotics & Automation Magazine*, vol. 25, no. 1, pp. 22–122, Mar. 2018.
[19] P. I. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 2nd ed., ser. Springer tracts in advanced robotics. Cham: Springer, 2017, no. 118.



Jörn Malzahn is a postdoctoral researcher at the Humanoid and Human Centred Mechatronics lab at the Istituto Italiano di Technologia in Genoa. He received his Dr.-Ing. degree from the Institute of Control Theory and Systems Engineering (RST) at TU Dortmund University in 2014. His research interest focuses on lightweight bio-inspired designs and control algorithms, to improve energy efficiency as well as strength of future robots, while simultaneously donating sensitive proprioceptive capabilities to them.

Wesley Roozing is a postdoctoral researcher. He received the B.Sc degree in Advanced Technology in 2011 and the M.Sc degree in Systems and Control (specialisation Robotics and Mechatronics) in 2014 from the University of Twente, The Netherlands. He received his Ph.D. degree from the Italian Institute of Technology (IIT) and University of Genoa in 2018. His research interests include energy-efficient actuation and control, compliant robotics, bio-inspired actuation, force control, and walking robots.

Nikos Tsagarakis received the D.Eng. degree in electrical and computer science engineering from the Polytechnic School of Aristotle University, Greece, in 1995, the M.Sc. degree in control engineering and the Ph.D. degree in robotics from the University of Salford, U.K. in 1997 and 2000, respectively. He is currently the Head of the Humanoid and Human Centred Mechatronics Lab at the Istituto Italiano di Technologia in Genoa.