Multibody System Dynamics (2023) 60:533–561
https://doi.org/10.1007/s11044-023-09937-1

RESEARCH

Exudyn – a C++-based Python package for flexible multibody systems

Johannes Gerstmayr[1]

Received: 14 March 2023 / Accepted: 8 September 2023 / Published online: 9 October 2023

Abstract
The present contribution introduces the design, methods, functionalities, and capabilities of the open-source multibody dynamics code Exudyn, which has been developed since 2019. The code has been designed for rigid and flexible multibody systems, with a focus on performance for multicore desktop processors. It includes script-language-based modeling and it is intended to be used in science and education, but also in industry. The open-source code is available on GitHub and consists of a main C++ core, a rich Python interface including pre- and postprocessing modules in Python, and a collection of rigid and flexible bodies with appropriate joint, load, and sensor functionality. Integrated solvers allow explicit and implicit time integration, static solution, eigenvalue analysis, and optimization. In the paper, the code design, structure, computational core, computational objects, and multibody formulations are addressed. In addition, the computational performance is evaluated with examples of rigid and flexible multibody systems. The results show the significant impact of multithreading especially for small systems, but also for larger models.

Keywords Multibody system · Simulation · Python · C++ · Open source · Flexible · Multithreading

1 Motivation

There is a long history of open-source codes for multibody system dynamics, see for example a paper on the open-source code MBDyn dating back already 20 years [19]. Recently, codes have evolved in the computer graphics and robotics areas [4, 31]. However, these codes are specialized to rigid bodies and contact and are usually based on kinematic trees, using a minimal-coordinates formulation. As with flexible bodies or compliant joints, minimal coordinates are not always (computationally) advantageous, and extension of such existing, specialized codes is time consuming. Flexible bodies may be as complicated as higher-order nonlinear rods or flexible bodies with model-order reduction, leading to very different sizes and complexities of computational objects, having single degrees of freedom (DOF) masses versus flexible bodies with more than 100 flexible modes. Thus, flexible bod-

✉ J. Gerstmayr
johannes.gerstmayr@uibk.ac.at

[1] Department of Mechatronics, University of Innsbruck, Technikerstr. 13, 6020 Innsbruck, Austria

ies are much harder to integrate into existing environments that are based on rigid bodies and kinematic trees.

The motivation for the development of a new multibody dynamics code, therefore, relied on an open design and developer friendliness, while still achieving sufficient performance. In general, the code is intended to be simple, readable, and extensible, and user interaction shall be as simple and safe as possible. Only when being performance critical is this path abandoned for efficiency. Advanced C++ implementation techniques, such as C++11 and higher, are only used in cases where the code can be shortened or simplified significantly and where readability does not suffer. For the same reason, the usage of external libraries has been limited such that no external basic linear algebra library is used throughout, as it would dictate parts of the structure or the interface. Nevertheless, besides the standard template library, external libraries are used for Python bindings, unit tests, graphics, parallelization, as well as sparse linear algebra. Furthermore, while most open-source projects are missing full and up-to-date documentation of interfaces as well as the theory behind the objects, the documentation is an integrated part of Exudyn, which is continuously developed together with the code.

There are powerful open-source codes that include already flexible bodies, such as Projectchrono[1] [21, 30], MBDyn [20], or HOTINT [5]. In the short term, the adaptation or integration of an existing software package is more efficient, while for the long-term development of new modeling and solution methods a new software package may be advantageous. Even though Exudyn has been started from scratch, some of the existing linear algebra and helper routines, which were also used in HOTINT, have been revised and adapted to the new project.

Efficient C++ implementation, as well as parallelization techniques, have been already applied to multibody systems, see [2, 12, 22], however, not for the case of moderately small and redundant systems, see Sect. 4.5 for further details. Furthermore, maximum computational efficiency, which can be achieved on GPUs nowadays [22], may not always be required for practical applications as compared to code extensibility, an intuitive modeling language, or user functions. A large amount of time is usually spent to create and check the correctness of multibody models, which is why the user interface – no matter whether it is graphical or text-based – deserves high importance. Recently, progress with artificial intelligence (AI) and large language models [1, 23] may change the need for graphical user interfaces in the future. AI models can already create correct Python or MATLAB code for simple dynamic models, while closed-source and proprietary formats are inaccessible to it. An essential requirement is to provide enough explanation and examples for the desired modeling language on open-source platforms. Hereafter, AI models can imitate the code's programming style and the user only needs to revise and validate.

The present paper is a revised and significantly extended version of the proceedings paper at IMSD2022 [6].

2 Introduction to Exudyn

Exudyn – flEXible mUltibody DYNamics – has been created by the author as a successor of the previously developed multibody code HOTINT – High Order Time INTegration [5]. The latter code had been designed as a pure time integrator, but not as a multibody code, and thus reached significant limitations of its extensibility. Exudyn [7] has been planned

[1] https://projectchrono.org.

as a research and education multibody code and has been developed and maintained at the University of Innsbruck since 2019. Exudyn integrates model scripting in Python,[2] efficient solvers, equation building, graphics visualization, and integrated documentation. In many C++ codes, a Python interface was added after the program had already reached a high level of development.[3] In contrast, Exudyn builds on the existence of a Python interface and is developed simultaneously on the C++ and Python sides.

The underlying C++ code focuses on an implementation-friendly but otherwise efficient design. Important parts of the code are parallelized using specialized multithreading approaches, see Sect. 4.5. The Python interface is mostly generated automatically and includes interfaces to important base classes and all data structures. To avoid untrackable errors during simulation, extensive parameter checking is performed during model creation that allows the identification of the errors' origins. However, direct access to C++ object functions via Python is restricted, because the full exposure of computational objects to Python would require too many checks of valid arguments and it would impede future changes in the C++ code.

Exudyn is hosted on GitHub [7] that includes all sources, test models, examples, and documentation, however, precompiled versions can be obtained conveniently via PyPI [8]. Exudyn version 1.6.0 – the core part including tests – currently roughly consists of 160,000 lines, of which 64% are C++, 13% Python modules, 11.5% definition and code generation, and 11.5% tests. Furthermore, it should be noted that 43% of the C++ code is automatically generated. The automatic code generation not only alleviates consistency between Python and C++ interfaces but also creates the reference PDF manual and online documentation. As a consequence, the definition of quantities, such as the stiffness of a spring-damper, is provided only once, while this information is passed to C++ and Python interfaces and the documentation, all of them being always consistent. On average, only 200 lines of manual code are needed per item, such as a rigid body or a marker, which is important for code maintainability and for the realization of larger code changes. The Exudyn V1.6.0 documentation, see the file theDoc.pdf [7], covers close to 800 pages, from which tiny parts have been adapted in the present paper. The documentation includes installation instructions, tutorials, description of entry points and interfaces, theory, and reference manual for every of the more than 80 items, such as objects, markers, or loads. The documentation also includes tracking of changes and bugs, which allows users to immediately discover new features. A more user-friendly version is available on the web, allowing full-text search.[4]

Coupling between C++ and Python is realized with the header file-based library Pybind11 [16], which enables a rich Python interface, and small code size. Using Python's `setuptools`, Exudyn can be built for Windows, Linux, and MacOS (although limited), and has even been compiled for ARM-based Raspberry Pi. Instead of starting with the definition of nodes, objects, and markers, there exist more than 100 examples and test models, which can be taken as a starting point to create new models [7]. Furthermore, there are tutorials in the documentation [7], as well as tutorial videos on YouTube [9]. Possible use cases range from 1-DOF-oscillators in teaching to 1,000,000-DOF finite-element or particle simulations in applications. Exudyn has been already used in several scientific publications and within industrial applications, just to mention Refs. [14, 15, 25, 27, 33].

While the core part of the code is written by the main author, many colleagues contributed to tests, formulations, specialized objects, and examples, see the acknowledgments and de-

[2] Restricting to Python 3.6 and higher; for information on Python, see https://www.python.org.

[3] It is noteworthy to mention that MBDyn already included a Python interface in 2007 [18].

[4] see https://exudyn.readthedocs.io and https://jgerstmayr.github.io/EXUDYN.

tails in the documentation [7]. The present paper will show details on the design, methods, functionalities, capabilities, performance, and examples as well as test results. The description given here can only be a brief introduction to the code, omitting significant parts of the functionality, while the full and continuously updated documentation is available on GitHub [7].

2.1 Design of Exudyn

The design goal of Exudyn has been a comparatively lightweight multibody dynamics software package for research, education, and limited industrial applications. Linking to other open-source projects shall be possible via Python. It shall include basic multibody dynamics components, such as rigid bodies and point masses, flexible bodies, joints, and contact. Models shall be scripted in Python, either within a list-like collection of bodies or joints or a highly parameterized model using for loops and external libraries. Additional Python utility modules shall enable convenient building and postprocessing of multibody models, shifting implementation of complex but not performance-critical functionality to Python. It shall further include explicit and implicit dynamic solvers, static and eigenvalue solvers as well as parameter variation and optimization methods. It shall be designed to be able to both efficiently simulate small-scale systems and large-scale systems with up to 1,000,000 unknowns, thus requiring sparse techniques, efficient memory usage, and parallelization ab initio. It shall include a 3D visualization to show modeling errors and to allow user-defined objects and solvers in C++ and Python. The system core shall be a redundant multibody dynamics formalism with constraints, but also including kinematic trees for efficient modeling of rigid-body tree structures (which may be solved with explicit solvers). Parallelization and vectorization shall be enabled through multithreaded parallelization.

Certain aspects are explicitly not considered in the current version of Exudyn, including methods for GPU utilization and a graphical user interface for creating multibody models.

As mentioned in Sect. 6, the creation of models as well as the start and evaluation of simulations is done using scripts entirely written in Python. Thus, the C++ module in the background acts as a state machine, which is driven by the user's script. A larger set of Python functions allows the user to understand the current state of the system, not only of the coordinates but also of the total system setup. A dense network of safety checks is integrated to allow the user to distinguish illegal input from system errors, which may occur, e.g., if modules are not fully completed.

3 Structure of the code

The code consists of a C++ core module denoted as exudynCPP and a collection of Python utility modules, see Fig. 1.

3.1 C++ core module

The C++ core is (currently) exclusively[5] linked to Python using the C++ tool Pybind11 [16]. Since Pybind11 is highly templated, which can cause large compilation costs[6] and impedes

[5]Linking to other high-level languages may be realized in the future with limited effort, as most Python interfaces are created automatically; currently, there is no straightforward way to use the C++ code without Python.

[6]Due to specific measures to reduce compilation costs of Exudyn, an Intel Core-i9 with 14 cores (28 threads) can perform multithreaded compilation and linking of Exudyn V1.6.0 in only 80 s.

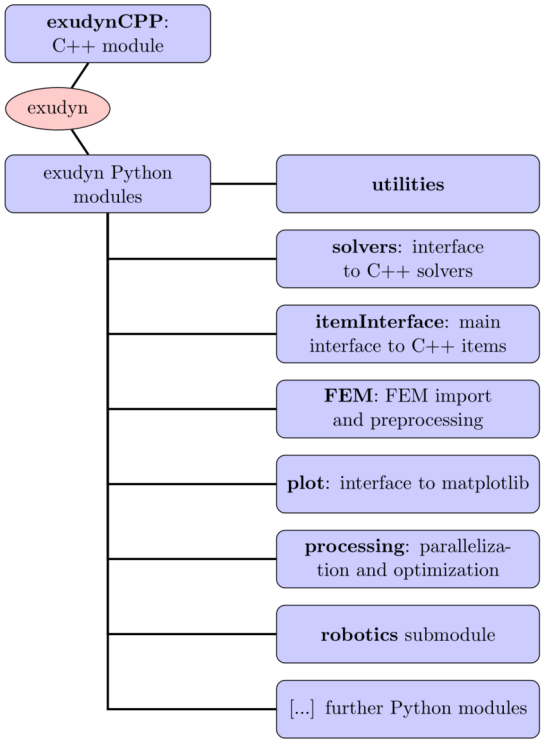

Fig. 1 Overview of Exudyn modules

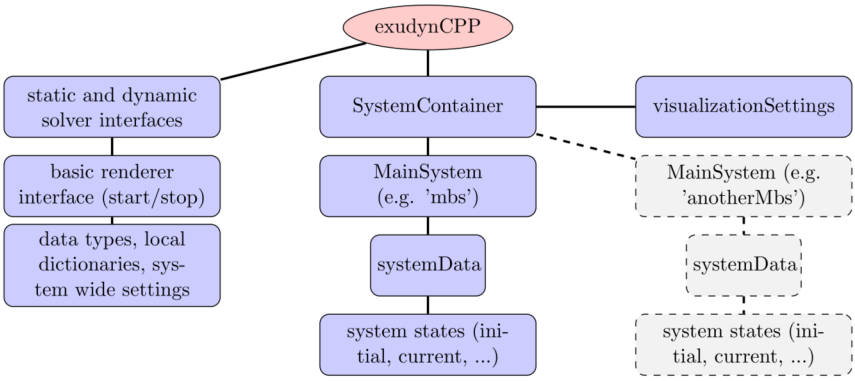

Fig. 2 Structure of Exudyn C++ module

conventional debugging, the interface between Python and C++ is reduced to core parts, see Fig. 2.

The C++ core parts include only a few structures, such as simulation settings, visualization settings, solvers, and sparse matrices, which are needed for efficient interaction with Exudyn. To reduce the interface size and improve error handling, there is a layer between the creation of items, e.g., a rigid body, marker, or sensor using conventional Python dictionaries. These dictionaries contain the whole data for a single item for reading and writing.

These data are checked before being transferring to the C++ module, which otherwise would lead to hard-to-interpret exceptions. Furthermore, as all items are transferred between C++ and Python by a single (Python) class interface, the change of the interface, as well as the implementation overhead, is greatly reduced. It is noteworthy that the Python class interface allows convenient type completion such that creating a certain body does not require knowing all parameters by their exact names.

Special emphasis has been laid on prechecking of parameter types and values, exception handling, and warnings to be an integral part of Exudyn. Since C++ errors are usually not interpretable and traceable for the user during solving, prechecks are performed in the Python interface itself.

3.2 Python modules

C++ code is computationally more efficient than pure Python code [32]. However, the maintainability of efficient C++ code is time consuming and, in general, error prone. Many parts of a multibody code, regarding model setup and validation, pre- as well as postprocessing are not performance critical. For this reason, these parts are shifted to Python modules. These Python modules, see also Fig. 1, are designed to make scripting as easy as possible and to reduce the C++ parts. Python modules include the interface (`exudyn.itemInterface`) between Python items and dictionaries transferred to the C++ core, utilities for optimization and parameter variation (`exudyn.processing`), and interfaces for rigid-body dynamics (`exudyn.rigidBodyUtilities`). In addition to many further modules for artificial intelligence, beams, and graphics, there is also a robotics submodule (`exudyn.robotics`) that can be used to easily set up serial or mobile robots.

3.3 Download, installation, and building

For more than two years, Exudyn has been hosted on GitHub [7]. The source code, which can be downloaded and compiled, contains the C++ code, setup files for easy compilation, the Python modules, examples as well as the documentation including its latex source files. Since Exudyn Version 1.2.0, it also has been added to the PyPI index [8]. This allows users to easily install precompiled wheels on many platforms, such as Windows, Linux, or MacOS using the simple command `pip install exudyn`.

Building[7] Exudyn is enabled with Python's `setuptools`. Within the main directory `exudyn/main`, the Python file `setup.py` includes all information to build Exudyn on many platforms using the command

```
python setup.py install --parallel
```

Successful building has been achieved for Windows 8, 10, and 11, Ubuntu (LTS) 18.04, 20.04, and 22.04, MacOS (BigSur) as well as for RaspberryPi 4b. However, not all of these platforms are maintained for compilation, which may require small adaptations.

4 Basic structure of the C++ core

The C++ core consists of a `MainSystem` that is used to add, modify, and read items (see Sect. 4.1) in the multibody system. Items are created by a typical object factory, which

[7] Compilation and linking of all C++ parts that are collected in a Python wheel together with all Exudyn Python libraries.

creates a C++ item from the Python dictionary definition. For interoperability between C++ and Python, every item has functions to read and write data structures.

For future extensibility and performance reasons, the items are separated into three layers:

- **Main** (M) layer: this is the top-level structure, which includes the interface to Python and further nonperformance critical parts. Python linking leads to higher compilation times, which is why this layer is only included in very few places. The main layer also includes preassembly checks.
- **Computational** (C) layer: this is the main, performance-critical layer. It is kept lightweight and is reduced to essential parts that are needed during computation. Focus is put on conciseness, reducing the number of lines for easy readability, and avoidance of programming errors. Typically, objects include only a few dozen up to 200 lines of code.
- **Visualization** (V) layer: this layer is only related to visualization and is thus decoupled from the computation. Visualization parts are only linked to graphics-related code, which is excluded if the code is compiled without the graphics renderer.

In addition to these layers, data structures for item parameters, such as the mass of a rigid body, are put into a separate `parameter` structure, which separates data and computational layers clearly. This should alleviate portability issues, especially to GPU-like hardware, in the future.

4.1 Computational items

The computational items are separated into a few conceptually different components, namely nodes, objects, markers, loads, and sensors. Separation into these groups makes implementation and extension easier and allows the implementation of various formulations. In addition to computational properties, items can be created and modified via the system they belong to. Furthermore, items can be visualized in the 3D rendering window. While nearly every item possesses the ability for some standard graphical representation, bodies can be enriched with conventional 3D graphics imported by STL[8] meshes. Naturally, all items are represented on every layer (M, C, V) through C++ classes, together with the corresponding Python interface class.

Markers provide interfaces on coordinate, position, or orientation levels that can be also used by loads, such as forces or torques. The separation of nodes and objects allows using a set of rigid-body nodes with different formulations for rigid or flexible bodies, leading to no additional implementation efforts. Joints are provided in index 3 and index 2 versions, which allows using solvers with index reduction and a convenient computation of initial accelerations. In addition to the rather independent implementation of objects or nodes, there is a specialized contact module with optimized contact search as well as evaluation of contact forces and Jacobians using multithreading. Many items allow integration of Python user functions, which break down performance, but allow nearly unrestricted interaction and couplings inside but also outside of Python (e.g., TCP connection with MATLAB). While it is recommended to simulate large-scale particle systems with one of the many excellent simulation codes such as Projectchrono [21], Exudyn is also capable of simulating and visualizing more than one million rigid bodies without special hardware or installation.

[8] Stereolithography file format; also known as Standard Triangle Language.

4.1.1 Nodes

Nodes provide the coordinates (and add degrees of freedom) to the system. They have no mass, stiffness, or other physical properties assigned. They can represent coordinates of different types, see Sect. 4.2.3, such as the position of a mass point, or generalized coordinates for minimal coordinates formulation.

Nodal coordinates and their assigned kinematical quantities (such as a rotation matrix) can be measured. Nodes can be used by a single object (e.g., rigid body), or shared by several objects, such as with finite elements. For the system to be solvable, equations need to be assigned for every nodal coordinate (which represents a system state).

4.1.2 Objects

Equations are provided by (computational) objects, and these equations are added up for each node. Generic objects or bodies need one or more associated nodes for the link to the coordinates, such as a position node for a mass point object. Connector objects provide equations via markers that transmit actions to the underlying coordinates. In the special case of algebraic constraints, the algebraic variables are automatically assigned to the constraints (without nodes), as a double usage of these coordinates is not considered meaningful.

Objects may provide derivatives and have measurable quantities (e.g., displacements or rotations) and they provide position or rotation Jacobians, which are used to apply, e.g., forces. While generic objects provide only equations, bodies also provide parts of the system mass matrix and they have a geometrical extension. When adding mass terms, the resulting mass matrix is identical if one adds a single mass point with mass m to a point node, or if two mass points with mass $m/2$ are added to the same point node. The same is true for a spring-damper if adding one spring with stiffness k or two springs with stiffness $k/2$ to the same markers leads to identical behavior (except for numerical round-off errors).

Objects can be as various as follows:

- **generic object**, e.g., represented by generic second-order differential equations;
- **body**: has a mass or mass distribution; markers can be placed on bodies; constraints can be attached via markers; bodies can be:
  - **ground object**: has no nodes;
  - **single-noded body**: has one node (e.g., mass point, rigid body);
  - **multinoded body**: has more than one node; e.g., **finite element**;
- **connector**: uses markers to connect nodes and/or bodies; adds additional terms to system equations either based on stiffness/damping or with constraints (and Lagrange multipliers). Possible connectors:
  - **algebraic constraint** (e.g., constrain two coordinates: $q_1 = q_2$);
  - **classical joint**, such as a revolute joint;
  - **spring-damper** or penalty constraint.

4.1.3 Markers

Connectors and loads cannot be directly attached to nodes or objects. Markers are interfaces for objects and nodes, such that connectors and loads can be attached to them. Markers can represent points with position Jacobian, a rigid-body frame with position and rotation Jacobians, or generalized coordinate(s). The main advantage of this approach is that connectors only need to be implemented for usage with specific markers, such as a rigid-body marker,

without knowing the exact behavior of the underlying objects, and with no difference if applied to a node or body, including any flexible body. The markers act similarly to what has been also denoted as virtual bodies [11]. As compared to other multibody codes, this technique alleviates the implementation of new objects, rigid-body nodes and connectors. None of them need to be reimplemented for different cases such as a spring-damper attached to a rigid body, a flexible body, or a node.

As an example, the marker of rigid-body type, e.g., `MarkerBodyRigid`, offers an interface that provides position, orientation, translational, and angular velocity to the connector to compute connector forces. In addition, the marker transmits Jacobians of translation and orientation with respect to the object's or node's coordinates. As the only requirement, these Jacobians need to be offered by the corresponding nodes or objects that allow rigid-body connectivity. As a final consequence, being a main difference from HOTINT, ground objects, and ground nodes can be easily realized as entities without coordinates (unknowns). They provide a spatially fixed position and orientation where markers can be attached in the same way as to rigid bodies. Jacobians for ground objects have zero columns as there are no actions transmitted to any coordinates. This design almost eliminates the need for any special cases in connectors.

4.1.4 Loads

Loads are used to apply (generalized) forces and torques to the system. Loads contain constant values, which are applied as a step function at the beginning of a dynamic simulation. If a static computation is performed prior to the dynamic simulation, it starts from the static equilibrium. The otherwise constant load values may be varied utilizing Python user functions, which may be evaluated in every call to the load, e.g., in Newton iterations.

4.1.5 Sensors

Sensors are only used to measure item quantities to create requested output quantities. They can be used to create closed-loop systems or they can be easily displayed using the `PlotSensor(...)` utility function, which provides an interface to Python's `matplotlib`.

4.2 System

The multibody system is represented by a `MainSystem`, which offers the Python interface via Pybind11 to a computational `CSystem` with all system-wide functions and a `VisualizationSystem` for all graphics operations, all of them implemented in C++ classes. Figure 3 shows the system graph of a double pendulum, highlighting the markers that are used as interfaces between bodies and joints or loads.

The computational system holds all data inside a data structure (`systemData`), consisting of:

- all system states (including some helpers such as accelerations);
- the lists of computational items, such as a list of nodes, a list of objects, etc.;
- local-to-global coordinate mappings;
- several lists of items with specific properties, such as a list of bodies and connectors, a list of objects with/without Python user functions, etc., which are used to reduce overheads in system functions, such as computation of a system mass matrix, and to facilitate parallelization.

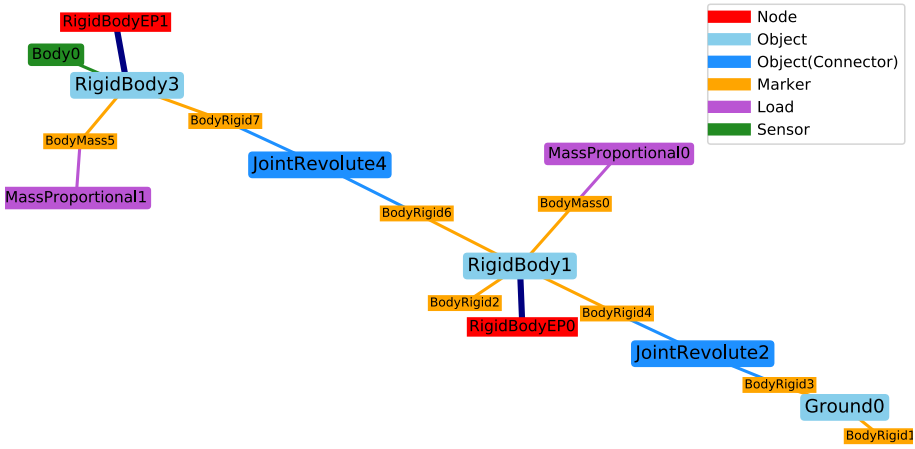

Fig. 3 System graph for double pendulum created with Exudyn's `DrawSystemGraph(...)` function. Numbers are always related to the node number, object number, etc.; note that colors are used for nodes, objects, markers, etc.; the label names (which do not include `Object..`, `Node..`) need to be interpreted with the color information (Color figure online).

System states are available for a list of configurations, mainly:

- current configuration (which is computed during or at the end of computational steps);
- reference configuration (defines state where joints are defined and used for finite elements);
- initial configuration (prescribed by the user in nodes or given system-wide)
- start-of-step configuration (for solvers);
- visualization configuration (representing the configuration to be rendered).

In particular, reference and initial configurations are distinguished. Reference values represent a specific configuration for finite elements in which their (undeformed) geometry is defined. Reference values are also used by joints in which global joint axes may be defined. On the contrary, initial values represent coordinates added to reference values (even for rotational parameters) and thus have a more abstract meaning. They are updated along computational steps and may be stored or loaded to retrieve a specific solution of the system.

4.2.1 Assembling

After the user has created and added all items to a multibody system, some linking and prechecking of items has to be performed by calling the `Assemble()` function. This function includes subfunctions with the following tasks:

- preassemble object initialization;
- check system integrity;
- assign nodal coordinates to global (system) coordinates;
- assign object coordinates to global (system) coordinates;
- precompute item lists (for efficiency);
- initialize system coordinates for initial and reference configuration based on user-defined nodal quantities;
- initialize the general contact modules;

– postassemble object initialization (e.g., precompute internal parameters that are not known at object creation).

During the system-integrity check, a large number of type and size checks are performed before assembling the model, such that common pitfalls are avoided for the user. These checks guarantee valid ranges of all indices in items, such as node indices in bodies or marker indices in connectors. These checks cannot be performed earlier, because in special cases – due to the cyclic dependency of objects – indices may be added at a step later than creation. Furthermore, items include a large amount of information, e.g., the valid type of markers for connectors, the type of nodes for objects, or available output variables in items.

Python user functions may even change the connectivity of objects or nodes. However, in such cases parts of the `Assemble()` function need to be called for the system to stay consistent. It is still possible that the user changes indices with low-level functions at a point where no checks are performed hereafter for efficiency reasons. Such cases are caught by exceptions within range checks in every array, vector, or matrix function.[9] The system can be visualized and quantities can be measured as soon as the assembly process has been completed. After adjusting any of the 400 simulation and visualization settings (or leaving defaults), solvers can be called for creating complex simulation environments, see the next sections.

4.2.2 Computational functions

The computational system `CSystem` is one of the larger parts of the C++ code, implementing widely parallelized system functions that are used within static and dynamic solvers as well as for system linearization. These functions include computation of the system mass matrix, constraint reaction forces, and generalized forces of objects and loads.

Hereby, components of the mass matrix are summed for all objects that are linked to certain nodal coordinates. Generalized forces for objects are summed on the so-called right-hand side (RHS) for respective nodal coordinates, independently of whether they represent applied forces of loads or internal (e.g., elastic) forces of elastic bodies. Solvers then make use of these system functions to compute residuals. Furthermore, Jacobians can be computed by the system (currently not in parallel). The system Jacobian is assembled from object Jacobians, using their internal mode of computation, which can be either numerical, analytical, or automatic differentiation. For verification, some switches exist to compare analytic Jacobians against their numerical counterpart.

4.2.3 Coordinates and local-to-global mapping

The difference between (local) computational objects and the (global) system is the mapping of local (unknown) coordinates to global coordinates. This mapping is denoted as local-to-global (LTG) mapping, which is essential both for the computation of object quantities, such as the position or deformation, as well as for the action of forces onto objects. In general, the LTG-mapping follows the node numbering, where system coordinates $\mathbf{q}$ consist of subvectors for nodes i with coordinates $\mathbf{q}_{node,i}$,

$$\mathbf{q} = [\mathbf{q}_{node,0}^{\mathrm{T}},\ \mathbf{q}_{node,1}^{\mathrm{T}},\ \ldots]\,. \tag{1}$$

[9] These range check exceptions cause a significant overhead of usually 20–30%, which can be removed by setting `sys.exudynFast=True` prior to the Python statement `import exudyn`.

The system includes first- and second-order *o*rdinary *d*ifferential *e*quations (ODE) together with algebraic equations, with respective coordinates. The set of available coordinate types is:

- coordinates for first-order differential equations (ODE1);
- coordinates for second-order differential equations (ODE2);
- coordinates for algebraic equations (AE);
- data (history) coordinates (Data).

Equation (1) holds for any of the above-mentioned coordinate types and assembles the four main system coordinate vectors $\mathbf{q}$ for ODE2, $\mathbf{y}$ for ODE1, $\boldsymbol{\lambda}$ for AE, $\mathbf{x}$ for Data. While it would be sufficient to have ODE1 coordinates instead of ODE2 coordinates, it can boost efficiency and simplify implementation using ODE2 coordinates with an associated mass matrix and known relations between position, velocity, and acceleration. ODE2 coordinates also include compressed coordinates for Lie-group nodes, allowing singularity-free computation of rigid bodies with 3 rotation parameters [14]. In addition to algebraic coordinates, there is also a need for data coordinates or history variables, which can store information on previous steps, not related to physical quantities, such as plastic strains, slip, or contact variables.

4.3 Functionality of the present code

The current Exudyn version 1.6.0 offers a considerable capability to solve rigid and flexible multibody systems. In particular, there exist standard rigid bodies and point masses, both in 2D as well as in 3D. As many applications are fully planar, special functionality exists for 2D bodies – in particular for performance reasons, while the general formalism is always 3D. Additionally, 1D objects can be used for simple mechanisms or to create linear or rotational motion just for one fixed axis.

Flexible bodies can be created using the floating frame of reference formulation. This formulation offers a reduced-order version, which can incorporate any modal reduction [33]. Import of mesh and modal reduction can be done with the included FEM module, which allows the straightforward realization of flexible mechanisms. Furthermore, external finite-element tools, such as NGsolve [29], are attached to enable direct coupling with large-deformation solid (tetrahedral) finite elements.

Large-deformation cable elements exist in 2D, also including special techniques for axially moving beams, while 3D rods (beam) are currently under development and possess restricted functionality.

In most cases, researchers will not find one of their required components, such as special flexible bodies or connectors. For this reason, there exist generic objects, such as `ObjectGenericODE2` or `ObjectGenericODE1`, which allow the creation of any mechanical system based on differential equations defined in Python. Furthermore, connectors can be extended or modified through Python user functions, e.g., to create special force laws, or for loads to obtain arbitrary time-dependent values. Python user functions[10] have the advantage that any change does not require recompilation and user functions can make use of (nearly) every Python package including scientific computing (e.g., SciPy) or artificial intelligence (e.g., TensorFlow).

[10]Note that in case of errors, user functions would throw up C++ exceptions with no further information, leading to painful debugging. Due to the excellent features of Pybind11 and some efforts in the Exudyn implementation, these exceptions are translated back to Python, such that the user receives the exception in relation to the definition of the Python user function including line numbers.

While Python user functions may be a factor of 100 to 1000 times slower than their C++ counterpart,[11] it may be a convenient alternative to modify the underlying C++ code. In addition to the inevitable overhead for Python function calls, pure Python and NumPy functions can make use of just-in-time compilation, e.g., using the Python tool Numba [17], which reduces the Python overhead solely to the interfacing between Exudyn and Python.

4.4 Solvers

To optimize performance, the implementation of system-specific solvers is inevitable. For the possibility to perform different analyses of multibody systems, several basic solvers are available:

- Explicit dynamic solver: for efficient solution of nonstiff problems with solely ordinary differential equations, e.g., rigid bodies under contact or kinematic trees.
- Implicit dynamic solver: for the solution of stiff or differential-algebraic equations.
- Static solver: to perform kinematic analysis or for computation of initial values for static equilibrium.

On top of these basic solvers, optimization or sensitivity analysis, shooting methods and numerical differentiation are available. Furthermore, the Jacobian of the dynamic implicit solver is used to obtain the linearization at a specific configuration, allowing to perform an eigenvalue analysis for this linearized system. Certainly, some advanced solvers are not implemented, such as an adjoint solver for the dynamic solution or special nonsmooth contact solvers.

Computational expenses for the computation of a single static or dynamic step are distributed across several computational functions of the `CSystem`. These functions mostly represent the residual of the equations of motion:

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \frac{\partial \mathbf{g}}{\partial \mathbf{q}^{\mathrm{T}}}\boldsymbol{\lambda} = \mathbf{f}(\mathbf{q}, \dot{\mathbf{q}}, t)\,, \tag{2}$$

$$\dot{\mathbf{y}} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^{\mathrm{T}}}\boldsymbol{\lambda} = \mathbf{f}_{ODE1}(\mathbf{y}, t) \quad \text{and} \tag{3}$$

$$\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{y}, \boldsymbol{\lambda}, t) = 0\,. \tag{4}$$

System-level computational functions are split into the computation of the state-dependent mass matrix $\mathbf{M}(\mathbf{q})$, the right-hand side $\mathbf{f}(\mathbf{q}, \dot{\mathbf{q}}, t)$ of ODE2 equations, the right-hand side $\mathbf{f}_{ODE1}(\mathbf{q}, \dot{\mathbf{q}}, t)$ of ODE1 equations, the residual of constraint equations $\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{y}, \boldsymbol{\lambda}, t)$, reaction forces $\frac{\partial \mathbf{g}}{\partial \mathbf{q}^{\mathrm{T}}}\boldsymbol{\lambda}$, and Jacobians of Eq. (2) with respect to the according static or dynamic solver. Note that the linear system of equations in the Newton method is solved either by a simplistic dense solver with little overhead and no memory allocation for small-scale systems or by the Eigen- [13] based `SparseLU` solver, using a supernodal approach for factorization of nonsymmetric sparse matrices. It should be mentioned that Eigen is only used for sparse systems, while dense linear algebra is fully implemented in Exudyn, as this leads to much shorter compilation times and simpler debugging. For large-system vector operations, multithreaded parallelization and Intel's advanced vector extensions (AVX) are utilized.

[11] On the Intel i9 (14 cores) test computer, the pure call to Python functions via Pybind11 leads to an overhead of approximately 1 μs, while C++ can call item functions in less than 10 ns. Costs increase accordingly if Exudyn interface functions are called within user functions.

To avoid reparameterization of rotation parameters in explicit dynamic simulations, a Lie-group formulation has been developed for Exudyn that can directly update rotation parameters using the rotation vector [14]. Solvers and rigid bodies (and in future beam elements) are adapted to special Lie-group nodes, which simplify kinematic operations and may lead to a constant mass matrix.

Specifically for large-scale models as well as for cases where many simulation runs are required (e.g., parameter variation or optimization), short CPU times are needed. Due to the large variety of problems, ranging from 1-DOF to 1,000,000-DOF problems, tuning of solvers is required and usually may influence the CPU times by a significant factor. Specifically, we mention switches between dense and sparse matrix computation, avoiding frequent calls to Python user functions, adjust Newton and discontinuous iteration parameters, e.g., switch to modified Newton. Whenever possible, explicit solvers should be used. Systems with purely constant mass matrices (e.g., using Lie-group nodes or ANCF elements) avoid the time-consuming rebuilding of the mass matrix. To determine the time-consuming parts of the solver, an integrated facility runs various timers and can show the amount of time spent, e.g., in Jacobian computation, file writing, or visualization, which allows the user to adjust the solver settings for performance improvements. For a detailed description also see the section 'Performance and ways to speed up' in the documentation, theDoc.pdf [7].

4.5 Multithreaded parallelization

Attempts for parallelization in multibody system dynamics have been made already by Anderson and Duan [2], however, only for specialized multibody formulations and solvers. Furthermore, González et al. [12] introduced a nonintrusive parallelization technique for multibody systems, however, stating that OpenMP is not appropriate in the case of small system sizes and showing absolute performance gains only around 15%. Another paper on parallelization of large-scale multibody systems refers to small systems in the introduction, suggesting low-level parallelization for this case [22] – very similar to what is used in Exudyn.

Therefore, a multithreading approach using C++ `std::threads` has been applied to Exudyn, which includes parallelized computation of mass matrix, right-hand sides (elastic forces and loads), reaction forces, and of all functions in the contact module. There exist two different task managers that are compared in this paper. The regular multithreading is taken from NGsolve [29], which includes a task manager for thread-based parallelization. Upon start, the task manager creates several threads, which continuously wait for new tasks to work with. Simple load management is used to balance tasks (such as the computation of a body's mass matrix) on the available threads.

A modified version, with minimum intertask communication and no load management (therefore suited for tasks with equal loading only), has been implemented and is denoted as the microthreading task manager. As will be shown in the examples, the microthreading approach can effectively parallelize very small systems. Note that most overhead for the task manager is due to the inevitable synchronization of atomic variables for all used tasks at the CPU level, needed at the beginning and end of a set of tasks and for load management. This synchronization time can be much larger than the computation time of the task itself, see the results of the mass–spring-damper system. Note that both task managers are tested within two different compilations of Exudyn, in which NGsolve [29] is the default one, also available with pip installers.

Parallelization may be applied to any 'for loop' at any computationally relevant function of the code, reading in the serial version (in pseudo C++ code):

```
for (int i=0; i < numberOfBodies; i++)
{
  const RigidBody& rigid = system.GetBody(i);
  const Vector& LTG = system.GetLTG(i);
  rigid.ComputeResidual(objectRHS);
  systemRHS.AddVector(objectRHS, LTG);
}
```

This function computes the object's RHS and adds this vector to the corresponding coordinates, defined by the LTG mapping in the system RHS. The parallelized version is realized via C++ lambda functions and called `ParallelFor(...)` [29]:

```
ParallelFor(numberOfBodies, [&system, &systemRHS, &objectRHS, &
    numberOfBodies](int i)
{
  const RigidBody& rigid = system.GetBody(i);
  const Vector& LTG = system.GetLTG(i);
  rigid.ComputeResidual(objectRHS);
  temp[GetThreadID()].AddVector(objectRHS, LTG);
})
systemRHS.AddVectors(temp);
```

In the parallel version, the 'for loop' runs independently for the number of threads defined during initialization. Therefore, data is written into thread-local vectors `temp[GetThreadID()]`, which are finally added to the system RHS. The final step is done in serial mode because solvers are currently only available in serial mode. The frequent call of a solver, as is done in the presented examples, would require an appropriate multithreading interface between Exudyn and the linear solver without starting or stopping threads. However, current open-source sparse direct solvers do not offer a suitable interface.

Finally, there is a special module for contact computations, allowing to efficiently compute contact between spherical particles (or clusters), triangulated surfaces, and beams. This module is decoupled from the otherwise item-based structure of the code and thus allows highly optimized computations using search trees and advanced parallelization techniques.

5 Advanced features

This section highlights some unique features that are designed to help the user to conveniently create multibody models.

5.1 Python interface

Models can be created not only by the Python interface but also by using many convenient functions of Python modules that are integrated on the Python side of Exudyn. These modules allow the creation of parameterized models, to add rigid bodies with inertia transformation, joint-axis computation or to add graphics annotations. High-level functions allow the creation of a reeving system with nonlinear finite elements or to set up a serial robot. Parameterized models follow naturally, which may be varied either manually with for loops, or automatically through advanced Exudyn Python functions, such as `ParameterVariation(...)`. Parameterized models can be directly optimized with `GeneticOptimization(...)` or `Minimize(...)`. A finite-element preprocessing tool allows to

import data from ABAQUS or Ansys or can use the NGsolve package [28, 29][12] by specialized interfaces. Well-known Python packages such as SciPy are integrated to compute eigenmodes, system linearization, optimization, or sensitivity analysis.

5.2 Automatic code generation

As part of the design, code duplication is avoided – for manually written code – as much as possible. This mostly affects data structures and interface functions. The more the data structures are open to the user, the easier and safer the usage becomes. As an example, being able to adjust every special solver setting, the usage becomes more transparent as compared to fixed values for special tolerances or iterations. Similarly, the evaluation of system or solver states allows the user to comprehend interrelations and debug problems.

Extending interfaces may be time consuming and programmers may prefer to hard-code new features first. Extensions always require specification of the interface structure, some documentation (e.g., about special behavior), range checks, a test case, and an example. Thus, extensions to the interface always involve a change to the program in several files, and an easy-to-extend mechanism for data structures reduces the mentioned problems in the long run.

As for the scalar mass of a rigid body, there are C++ read and write functions, the data variable for mass, the Python interface function (C++ and Python), the range checking function, and the documentation. This gives at least seven places at which code needs to be created (and documented). In the current implementation, for interfacing only, a regular object data variable is repeated 21 times in the C++ and Python code. All of these code parts are autocreated from a single definition, which allows to change or adapt such interfaces easily. It also avoids that some of these interface functions would be missing or wrong and untested.

5.3 Graphics

Open-source software in dynamics often focuses on core functionality, while graphics or importing functions are missing. While this may be established scientific practice, it means a greater potential for users to make errors during model generation, which can only be discovered after extensive searching.

Exudyn, therefore, has a 3D visualization as an integrated feature, similar to what has been available in HOTINT [5]. In this way, general geometric properties and errors in the kinematic behavior can be detected instantaneously. The highly integrated graphics renderer is based on the multiplatform, but otherwise tiny tool GLFW.[13] The graphics core still uses old-style OpenGL 1 standards, which is sufficient to render 3D models with excellent frame rates and allows light as well as shadow adjustments through visualization settings. In the graphics window, items can be verified by mouse-clicking and properties may be checked. Integrated graphics is convenient for interactive and reactive models in teaching but also let students create models within a short time.

6 Create a simple example in Exudyn

This section is intended to provide a simple but fully functional example in Python. Requirements are an installed Miniconda with NumPy and Matplotlib installed or a full Ana-

[12] https://ngsolve.org.

[13] *G*raphics *L*ibrary *F*rame*w*ork: https://www.glfw.org.

conda[14] (recommended). The example runs on a wide range of Exudyn and Python versions (3.6–3.10) and has been tested with Exudyn V1.6.0. The example models a rigid pendulum, using a redundant coordinate formulation based on a rigid body (using Euler parameters) and a generic joint, restricting all relative translations and rotations except for rotation around the local z-axis:

Most of the example script should be self-explanatory based on the given comments. The example follows the standard Exudyn script structure, as follows:

1. import `exudyn`, `exudyn.utilities` and other required packages;
2. create a system container and a mbs system, to which all items are added;
3. add bodies (in the more general case, first nodes are added, then objects);
4. add markers on bodies or nodes, providing interfaces for joints or loads;
5. add joints or connectors (spring-dampers, etc.);
6. add loads and sensors;
7. assemble mbs;
8. define simulation and visualization settings;
9. start simulation of mbs.

Optionally, as shown in the following example script, the 3D renderer may be started before simulation, to show the updated state of the system during simulation.

Example script:

```
#import exudyn modules and numpy:
import exudyn as exu
from exudyn.utilities import * #imports main sub-libraries
from exudyn.plot import PlotSensor

#create empty multibody system
SC = exu.SystemContainer()
mbs = SC.AddSystem()

#parameters:
g = [0,-9.81,0] #gravity
L = 1           #body dimensions
b = 0.1         #body dimensions

#create inertia and body graphics
iCube = InertiaCuboid(density=5000, sideLengths=[L,b,b])
graphics=GraphicsDataRigidLink(p0=[-0.5*L,0,0],p1=[0.5*L,0,0],
            axis0=[0,0,1], radius=[0.5*b,0.5*b],
            thickness=b, width=[b,b], color=color4red)

#add rigid body (in background creates node and object)
[n0,b0]=AddRigidBody(mainSys=mbs, inertia=iCube,
                     nodeType=exu.NodeType.RotationEulerParameters,
                     position=[L*0.5,0,0], gravity=g,
                     graphicsDataList=[graphics])
#ground body and markers
oGround = mbs.AddObject(ObjectGround())
markerGround=mbs.AddMarker(MarkerBodyRigid(bodyNumber=oGround,
                                localPosition=[0,0,0]))
markerBody0J0 = mbs.AddMarker(MarkerBodyRigid(bodyNumber=b0,
                                   localPosition=[-0.5*L,0,0]))
#revolute joint (free z-axis), attached to markers
```

[14] https://www.anaconda.com.

```
mbs.AddObject(GenericJoint(markerNumbers=[markerGround,
              markerBody0J0],constrainedAxes=[1,1,1,1,1,0],
              visualization=VObjectJointGeneric(axesRadius=0.01,
                                                axesLength=0.1)))
#add sensor which can be plotted or evaluated:
sPos = mbs.AddSensor(SensorNode(nodeNumber=n0, fileName='pos.txt',
              outputVariableType=exu.OutputVariableType.Position))

#finalize mbs, prepare for visualization and simulation:
mbs.Assemble()

#settings (out of several 100 options, using defaults):
simulationSettings = exu.SimulationSettings()
simulationSettings.timeIntegration.numberOfSteps = 1000
simulationSettings.timeIntegration.endTime = 4

#start visualization and solver:
exu.StartRenderer()
mbs.WaitForUserToContinue()
exu.SolveDynamic(mbs, simulationSettings)
exu.StopRenderer() #safely close rendering window!

#plot result of sensor 'sPos'
PlotSensor(mbs, sPos, closeAll=True)
```

The graphical output of this example can be seen in Fig. 4. When pressing space, simulation finishes in less than 0.05 s and the output of the sensor is shown. For more detailed examples, see [7].

7 Performance tests and evaluation

The present section evaluates seven different small to medium-sized multibody examples to demonstrate speedup due to multithreading. If not otherwise mentioned, the tests were performed on a desktop workstation PC with one Intel Core i9-7940X CPU with 14 cores (supporting 28 threads via hyperthreading), 3.10 GHz, 96 GB RAM, Windows 10 that is denoted as 'Intel i9 (14 cores) test computer' in the following. Note that regular runs on other CPU architectures show similar behavior as reported here, including mobile, AMD, and Apple M1 processors, while performance on Linux is better, in general. As it is well known that hyperthreading does not improve performance if the underlying code is highly optimized (using all floating-point units or reaching memory-bandwidth limits), the performance is usually only shown for the number of threads set to the number of available cores, as performance does not increase with more threads.

The overall performance of the code is not specifically compared with other codes within these examples. However, to give an idea, the CPU time of the stiff flyball governor[15] of the IFToMM multibody benchmarks is used and compared for a similar implicit trapezoidal integration method and fixed step size of $5 \cdot 10^{-4}$ s. The comparison on comparable platforms and CPUs, using only serial computation, shows:

– MbsLab (IFToMM reference): Intel Core i7-4790K@4.00 GHz, 8 GB RAM, Windows 8.1 Pro, 64 bit: 0.853 s CPU time;

[15] https://www.iftomm-multibody.org/benchmark/problem/Stiff_flyball_governor.

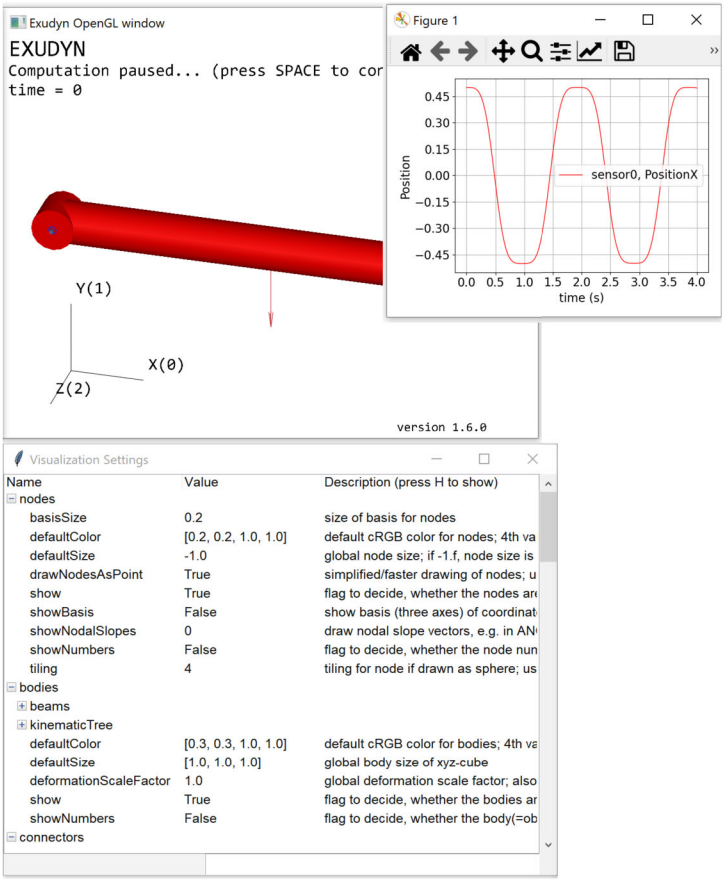

Fig. 4 Screenshots of 3D visualization in Exudyn: graphical representation of simple rigid-body example (top left), visualization settings (bottom), and `PlotSensor(...)` output (top right)

– Exudyn / minimal coordinates: Intel i9 (14 cores) test computer running on UBUNTU: 0.426 s CPU time;
– Exudyn / redundant coordinates: Intel i9 (14 cores) test computer running on UBUNTU: 0.908 s CPU time.

This comparison, which has been computed only on a single core in both cases, shows the good overall performance of Exudyn, being approximately two times faster.[16] While Exudyn is a general rigid-flexible simulation code, the results also demonstrate the significant differences between two diverse multibody formalisms. The fully redundant formulation leads to 28 nonlinear ODE2 equations and 28 nonlinear algebraic constraints, which are solved within approximately 75,000 iterations of a modified Newton scheme, where most time is spent on computation of constraint reaction forces and solving the linear equations in Newton. In the minimal coordinates (kinematic tree) case, the system has only four equa-

[16] As both processors are different, we would like to mention that the i7 processor is 3 years older, comparing launch dates of the two processors. However, the turbo frequency of both CPUs is 4.40 GHz, while the multithreading performance as well as memory clock rates are better on the i9 processor.

tions of motion, which are highly coupled. Most of the time is spent on the computation of the spring-damper forces as they require computation of the full tree kinematics for each connection point – with potential for optimization in the future.

Further tests focus on the performance using multithreading parallelization for multicore CPUs that are available on supercomputers, laptops, but also low-cost computers. To alleviate test evaluations, Python 3.7 has been used for the microthreading approach and Python 3.9 for the regular multithreading approach. As test computations run almost exclusively at the C++ side of Exudyn, there is no significant difference between Python versions having equal CPU, platform, and compiler. In all tests, the solver is using sparse matrices, except for the mass–spring-damper system with less than 20 unknowns, which switches to dense matrices in this case. For implicit time integration, the modified Newton method is used, which only updates Jacobians in case of slow convergence. Furthermore, visualization, as well as file output, is turned off and the computer is running almost idle.

7.1 Mass–spring-damper system

The first example is a one-dimensional (1D) chain of n mass–spring-dampers, with mass $m = 1$ kg, viscous damping $d = 2$ N m/s and spring stiffness $k = 800$ N/m and the first spring fixed to ground. The system consists of n mass points, modeled with 1D nodes (having only one unknown), and a spring-damper, which acts on the coordinates of two adjacent nodes. There is only one scalar, linear equation of motion for every mass and spring-damper, such that most of the computational efforts are overheads within the system calls. We simulate 100,000 time steps over 100 s of simulation time, which results in less than 0.1 s of computation time for the single-threaded one-mass system using (implicit) generalized alpha method [3]. Due to speed measurements, some overheads add up, which otherwise would lead to considerably less than 0.1 s of simulation time.

As in all examples, two multithreading approaches (regular and microthreading) are compared. The general overhead for multithreading, independently of the number of masses, is measured. For microthreading, the general overhead is almost independent of the number of threads and measured as ≈ 3.5 μs per step. For the regular multithreading, the general overhead is ≈ 1 μs per thread and step. Additional overheads due to load balancing add up in the regular multithreading, measured as ≈ 4 μs per step. For a larger number of masses load balancing helps to achieve better results for the regular multithreading (therefore it is used by default in Exudyn). Note that these times are valid for the Intel i9 (14 cores) test computer and may be different on other hardware.

The CPU times for different numbers of masses, threads, and task managers are shown in Figs. 5 and 6. For 10,000 masses, the maximum speedup is 4.59 using 12 threads with the microthreading approach, and the maximum speedup is 4.67 using 14 threads with regular multithreading. Furthermore, the break-even for 4 threads is at 27 masses in microthreading and at 33 masses for regular multithreading. Slightly worse, the break-even for 12 threads is at 32 masses in microthreading and at 70 masses for regular multithreading.

7.2 Robot: redundant coordinates

A more relevant multibody test is using n_m serial manipulators with 6 DOF each, see Fig. 7. In the first test set, redundant coordinates and constraints are used. This gives $6 \times n_m$ rigid bodies (using 4 Euler parameters for rotations), and $6 \times n_m$ constraints, leading to $42 \times n_m$ ODE2 coordinates per robot as well as $42 \times n_m$ constraints. Due to a generic joint approach, $6 \times n_m$ constraints act on the Lagrange multipliers for the free rotation axis, being always

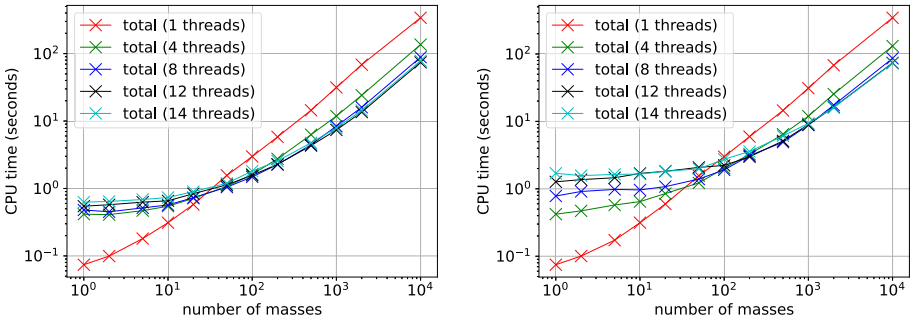

Fig. 5 CPU times versus number of masses in the mass–spring-damper system for different multithreading approaches on an Intel i9 (14 cores) test computer using a logarithmic scale; Microthreading (left) and regular multithreading (right); (Color figure online)

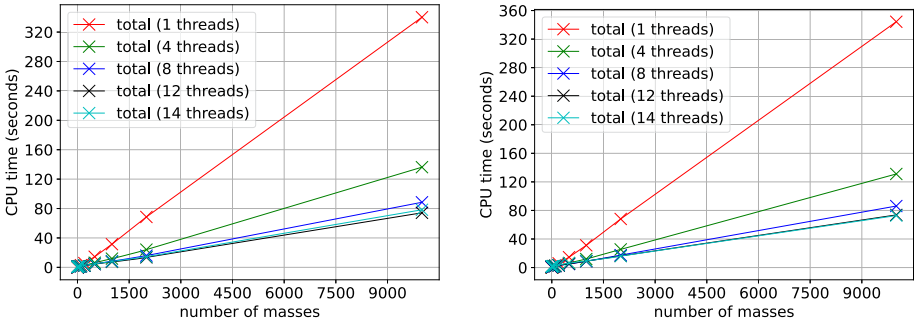

Fig. 6 CPU times versus number of masses in the mass–spring-damper system for different multithreading approaches on an Intel i9 (14 cores) test computer; Microthreading (left) and NGsolve regular multithreading (right); (Color figure online)

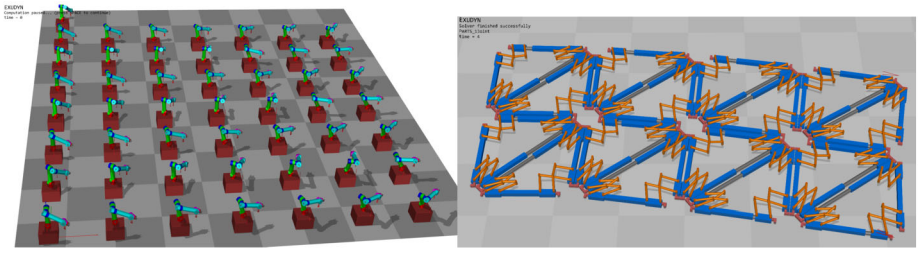

Fig. 7 Screenshots of test cases in Exudyn. Left: test case with serial manipulators; the number of manipulators is varied in this test, while 50 robots are shown exemplary; right: test case with $4 \times 2 \times 2$ cellular robots

zero. Furthermore, the drives of the robots are simple PD-controllers realized as spring-dampers with prescribed motion. Note that all robots move on different trajectories to avoid an overly simple structure of the problem. Even though the robots are uncoupled, the simulation of a factory with partial interaction of robots would experience similar computational

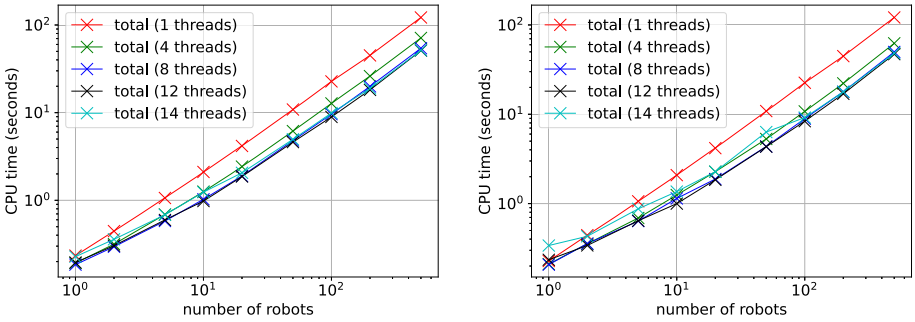

Fig. 8 CPU times versus number of robots with redundant coordinates for different multithreading approaches on an Intel i9 (14 cores) test computer using a logarithmic scale; Microthreading (left) and NGsolve regular multithreading (right); (Color figure online)

costs. The implicit generalized alpha solver is used to solve the system for 1 s simulation time with 1000 steps.

In this test case, the number of robots is varied and CPU times are measured for different numbers of threads, see Fig. 8. Both cases lead to nearly identical parallelization effects, while microthreading performs better for a few robots and a large number of treads. Note that in the case of microthreading already one robot sees little parallelization speedup with only 6 bodies involved. The maximum speedup with 12 threads and regular multithreading is 2.69 for 100 robots, while for all cases with 50 robots or more, the speedup stayed above 2.4 for 8 and 12 threads. Note that for 500 robots the system has 21,000 unknowns to be solved. Thus, the Jacobian computation and the solver cause already 25% of computation time in the serial version, causing a lower speedup factor.

7.3 Robot: minimal coordinates formulation

In a variation of the previous test, the system of robots is modeled with minimal coordinates. This leads to only $6 \times n_m$ unknowns in the system and no constraints. Furthermore, an explicit 4th-order Runge–Kutta time integrator is used to solve the system. The system is again solved for 1 s simulation time with 1000 steps.

In this test case, the number of robots is varied and CPU times are measured for different numbers of threads, see Fig. 9. Both cases lead to nearly identical parallelization effects, while microthreading performs better for a small number of robots and a large number of threads, see Fig. 9. The maximum speedup with 12 threads and microthreading is 6.90 for 500 robots (regular multithreading: 6.29), showing the excellent scaling of the explicit solver, for the case where no factorization and Jacobian are required.

7.4 Cellular robot PARTS

A rigid-body example is studied, which represents a highly parallel mechanism modeling cellular robots known as PARTS [26]. The system consists of triangular cells with actuators at each triangle side and special six-bar-linkages at the vertices.

The system with 16 triangles as shown in Fig. 7 (right) consists of 336 (planar) revolute joints, 288 planar rigid bodies, and 1641 total unknowns to be solved in every step of the implicit generalized alpha integrator. A second system of double size with 32 triangles and

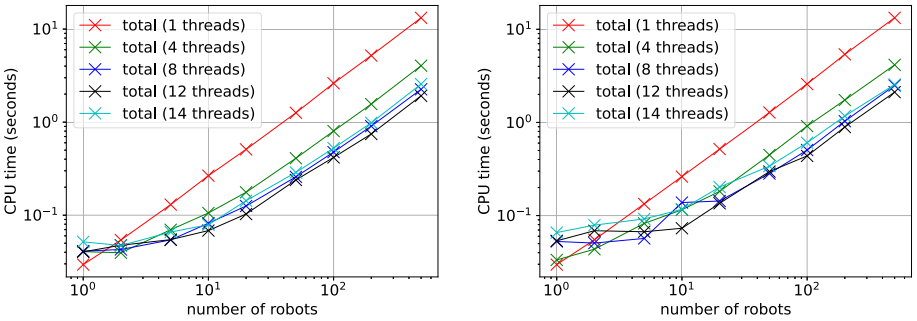

Fig. 9 CPU times versus number of robots with minimal coordinates for different multithreading approaches on an Intel i9 (14 cores) test computer using a logarithmic scale; Microthreading (left) and NGsolve regular multithreading (right); (Color figure online)

Table 1 CPU times for simulation of PARTS for 20 s of simulation time and microthreading; Intel i9 (14 cores) test computer

Number of Threads	Rigid Bodies	Time Steps	1	4	8	10	12
CPU time (s)	288	2000	8.00	3.20	2.42	2.23	2.10
CPU time (s)	576	1000	7.98	3.13	2.27	2.20	2.20

576 rigid bodies is investigated, as well. The CPU times for both systems are shown in Table 1, leading to a maximum speedup factor of 3.8 for the smaller system using 12 threads.

7.5 Belt drive

The first flexible multibody system represents a belt drive similar to the problem described in [24]. The belt drive consists of two pulleys under contact with a belt discretized with 480 ANCF elements following the formulation of [10]. The total number of coordinates including the 2D rigid bodies of the sheaves is 1926.

The system is solved for 0.5 s with 5000 time steps and constant step size. In the serial version (1 thread), most computational time of the implicit trapezoidal rule results in the evaluation of the elastic forces and the contact law between beam elements and pulleys (71.9%), nonlinear iterations related to contact (7.6%), and the backsubstitution for the Newton increments (9.4%). Jacobian evaluation and factorization take 10.8% of CPU time. Since the mass matrix is constant for ANCF elements, its evaluation time is negligible. The small time step leads to only 325 updates of the Jacobian with 43,016 iterations in total for the modified Newton method. Note that a simulation with the full Newton method would result in less than half of the Newton iterations, however, with a significant increase of computation time due to Jacobian evaluation and factorization.

The speedup due to multithreaded computation is shown in Table 2, giving a maximum speedup factor of 3.03 for 12 threads. The limited speedup is caused by the increasing amount of factorization and Jacobian computation costs (computed serial), while the evaluation of elastic forces shows speedups of 6.5. As can be observed in Table 2 as well as in previous tests, the optimal speedup is located around 12 parallel threads. The latter table also includes results with more than 14 threads, showing the poor performance of hyperthreading in this case.

Table 2 CPU times for belt drive with implicit time integration and regular multithreading; Intel i9 (14 cores) test computer

Number of Threads	1	2	4	6	8	10	12	14	28
CPU time (s)	97.8	65.2	44.7	37.9	34.9	32.8	32.3	32.7	73.0

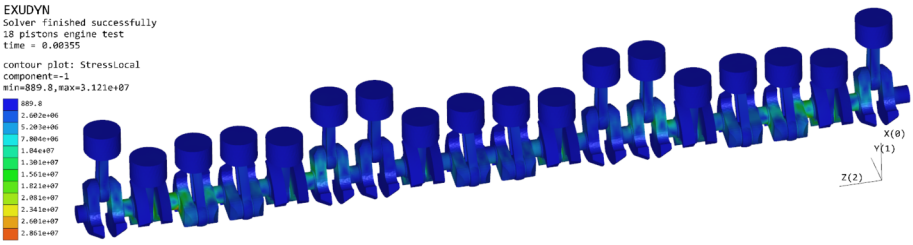

Fig. 10 Test model of an 18-piston engine with 37 flexible bodies

Table 3 CPU times for piston engine with implicit time integration, Python 3.9, Exudyn 1.6.0, regular multithreading, Intel i9 (14 cores) test computer

Number of Threads	1	2	4	6	8	10	12	14
CPU time (s)	76.7	44.1	30.0	23.2	19.9	18.8	17.8	19.1

7.6 Piston engine

A comparatively large model is tested based on a deformable 18-piston engine, see Fig. 10. All 37 bodies are flexible bodies using quadratic tetrahedral elements and the floating frame of reference formulation as published in [33]. The bodies include a total of 368,391 unreduced nodal coordinates. The Python package NGsolve [29] is used to mesh the flexible bodies and to export mass and stiffness matrices for each body.

The original DOF are reduced with the help of Exudyn's `FEM` module for every body using free-free eigenmodes, even though other modal reduction methods may be more accurate. The total time for preprocessing including geometry creation, meshing, mode computation, and computation of stress modes only takes 72 s CPU time, whereof eigenmode computation within a special Exudyn-NGsolve Python functionality takes 38.8 s. This easily allows performing parameter variations for the complete flexible multibody system.

The simulation is performed for 0.05 s with 2000 constant time steps, resulting in the CPU times given in Table 3 with a maximum speedup factor of 4.3 for 12 cores. This example again demonstrates the high potential of parallelization even in the case of a small number of bodies.

7.7 Parameter variation

In engineering applications, parameter variations play a larger role. Having an Exudyn model scripted in Python, only a few adaptions are required to perform a parameter variation using the simple function `ParameterVariation` from `exudyn.processing`.[17]

[17] Note that the variations for the current paper have made use of this library, however, only in serial mode to measure the according CPU times.

Table 4 CPU times for particle dynamics, LEO5 supercomputer

Number of Threads	1	2	4	8	12	16
CPU time (s)	7142	3865	2152	1319	991	861
speedup factor	–	1.85	3.32	5.41	7.21	8.30

Number of Threads	24	32	40	48	64
CPU time (s)	711	651	611	600	552
speedup factor	10.05	10.97	11.69	11.90	12.94

The Python core library `multiprocessing` is used to perform the parameter variations also multithreaded, allowing to compute a set of variations in parallel. As a simple test, 100 variations of the joint stiffnesses of the piston engine have been performed on the Intel i9 (14 cores) test computer, each of the variations running on a separate thread. While a single-threaded run takes 76.7 s, see Table 3, plus 18.5 s for loading the finite element data, the 100 runs only need 1010 s with 14 parallel threads (one thread per core) and they need 855 s with 28 parallel threads (hyperthreading). This gives a maximum speedup factor of 11.1 compared to 9520 s in serial computations. In cases with fewer overheads for loading data structures, speedup factors of 50 have been measured for 80 cores on a supercomputer.

7.8 Particle dynamics and supercomputer performance

The final example is used to show the potential for the current multithreading approach concerning a larger number of rigid bodies. The particles dynamics example is run on the local university's supercomputer LEO5, which is a distributed memory CPU and GPU cluster, having 62 compute nodes with 2×32-core Intel Xeon 8358 (Icelake) CPUs each with 2.60 GHz base frequency. The tests in the present section are performed on only one single node of the supercomputer, blocking all 64 cores also for test cases with the number of threads less than 64, to have minimum interference with other computations on the supercomputer.

Table 4 shows the time spent for the test model shown in Fig. 11 consisting of 100,000 rigid-body particles under contact and friction. A search tree (boxed search) with 13,824 fixed tree cells is utilized for optimized contact search. The step size is $1 \cdot 10^{-4}$ s and the total simulation time is 2 s, corresponding to 20,000 time steps. The explicit Euler method is used for time integration. As can be seen in Table 4, the maximum speedup factor is close to 13 for 64 threads, while more economically, only 24 threads are necessary to achieve a speedup factor of 10. It is worth noting that the chosen multithreading approach is limited by the CPU's memory bandwidth, such that the speedup factor for systems of this size and larger is less than for systems that fit into the cache.

7.9 Special notes on the use of Python

The close integration into Python has many advantages. There is a large number of standard Python packages, within linear algebra (NumPy), scientific computing (SciPy), symbolic computation (SymPy), and visualization (Matplotlib). On top of that, there are state-of-the-art machine-learning packages such as PyTorch and TensorFlow. They all can be directly

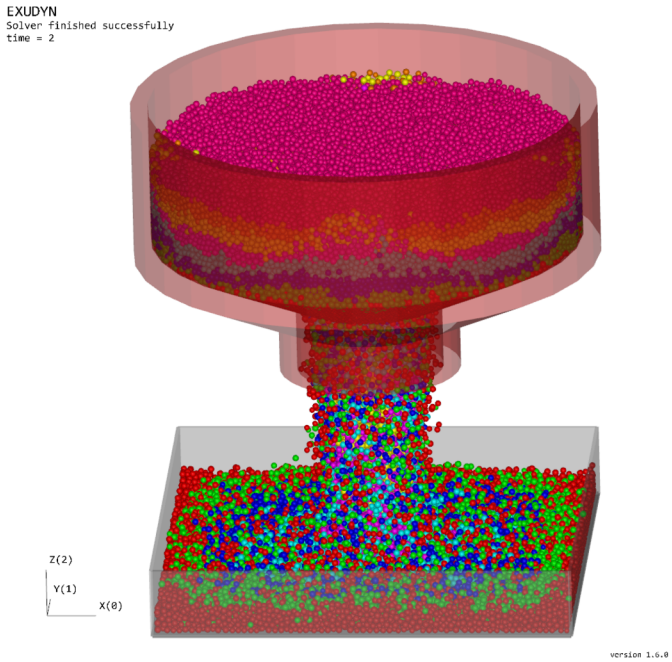

Fig. 11 Screenshots of test model for particle dynamics after 2 s; computations were performed on the supercomputer, while the screenshot was made with Exudyn running on Windows

connected to Exudyn, e.g., by using `stable-baselines3` reinforcement learning algorithms. Using ROS (or ROS2), we can also integrate Exudyn models into the robotics world. We can either perform realistic robot simulations or use a model to compute a prediction of a real robot or to create a virtual twin.

The Python language offers various interfaces to other languages. The language Julia,[18] which became popular in scientific computing, can integrate Python and access Exudyn functions and structures directly, allowing data exchange within a simple line of Julia code.

The open-source character and the online documentation of Exudyn allow training large language models (AI models) to write Python code for Exudyn almost automatically. In this way, ChatGPT [23] could already create simple models without interaction, making some significant errors, but mainly correct code. The errors made by such AI models show also the amount of intuitiveness the code has. Nevertheless, a larger number of examples allows future AI models to alleviate the creation of complex multibody models in the future and it may be the preferred way as compared to classical user interfaces.

The close integration into Python also has some disadvantages. Currently, there is no direct option to use Exudyn in pure C++, which could be useful for integration into embedded systems. Furthermore, Python performs a rather fast updating policy. The first Exudyn version in 2019 had been created for Python version 3.6, while now Python 3.10 is a standard and compilation is already possible on Python 3.11. The frequent updating of Python requires maintaining a larger set of different versions, at least for Windows, Linux, and

[18] https://julialang.org.

MacOS, giving 18 precompiled installers on PyPI [8]. Even though installers can be automatically generated, a large amount of data is generated that needs to be tested.

8 Conclusions and outlook

A new general-purpose C++ flexible multibody dynamics code has been made open source. The paper introduces the design and specific features of the code. In particular, computational performance is evaluated for multithreaded parallelization using novel threading approaches adapted for very short tasks. Tests reveal that parallelization is effective already for 50 degrees of freedom in very simple mass–spring-damper systems, with a speedup factor with implicit solvers typically from 3 to 5 with 12 cores. In the case of explicit integration, especially for larger-particle systems, speedup factors range from 7 on a regular workstation up to 12.9 at one node of a supercomputer. Furthermore, in more advanced rigid and flexible multibody systems, parallelization significantly reduces CPU time with as little as 12 rigid bodies. Even if speedup factors may seem to be too small as compared to those known from special linear algebra libraries, it shall be noted that the speedup takes effect for any new object in Exudyn, allowing to put less effort into the ultimate optimization of every single object.

Future developments will focus on parallelized solvers, as factorization and backsubstitution currently cannot run in parallel with the available SparseLU solver of Eigen [13]. While the code has made significant progress since its foundation in 2019, there is still a need for large deformation 3D beam elements, improved computation of Jacobians, and an improved contact solver, which will be the subject of future research and development.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s11044-023-09937-1.

Acknowledgements Exudyn could not have reached its current state without significant help from many colleagues. The team at the Department of Mechatronics is gratefully acknowledged, especially for supporting computer infrastructure and testing. In particular, contributions to robotics (Peter Manzl, Martin Sereinig), Lie-group methods (Stefan Holzinger), nonlinear beams and testing (Michael Pieber), and the floating frame of reference formulation (Andreas Zwölfer, TU Munich). Joachim Schöberl (TU Vienna) and his group contributed significantly to the parallelization and the coupling of the finite-element code NGsolve. Further acknowledgments are mentioned at GitHub [7]. Thanks to all!

The computational results presented here have been achieved in part using the LEO HPC infrastructure of the University of Innsbruck.

Author contributions Johannes Gerstmayr wrote all parts of the manuscript and computed all results.

Funding Open access funding provided by University of Innsbruck and Medical University of Innsbruck.

Declarations

Competing interests Johannes Gerstmayr is member of the Editorial Advisory Board of Multibody System Dynamics.

References

1. Ananthaswamy, A.: In AI, is bigger better? Nature **615**, 202–205 (2023)
2. Anderson, K., Duan, S.: A hybrid parallelizable low-order algorithm for dynamics of multi-rigid-body systems: part I, chain systems. Math. Comput. Model. **30**(9), 193–215 (1999)
3. Arnold, M., Brüls, O.: Convergence of the generalized-α scheme for constrained mechanical systems. Multibody Syst. Dyn. **85**, 187–202 (2007)
4. Coumans, E.: Bullet physics simulation. In: ACM (Association for Computing Machinery) SIGGRAPH 2015 Courses, SIGGRAPH'15, New York, NY, USA (2015)
5. Gerstmayr, J.: HOTINT – a C++ environment for the simulation of multibody dynamics systems and finite elements. In: Arczewski, K., Fraczek, J., Wojtyra, M. (eds.) Proceedings of the Multibody Dynamics 2009 Eccomas Thematic Conference (2009)
6. Gerstmayr, J.: Exudyn - a C++ based Python package for flexible multibody systems. In: Proceedings of the 6th Joint International Conference on Multibody System Dynamics and the 10th Asian Conference on Multibody System Dynamics 2020 (IMSD2022), New Delhi, India (2022)
7. Gerstmayr, J.: Exudyn GitHub repository. University of Innsbruck. https://github.com/jgerstmayr/EXUDYN (2023). Accessed 8 Mar 2023
8. Gerstmayr, J.: Exudyn, Python Package Index (PyPI). https://pypi.org/project/exudyn (2023). Accessed 8 March 2023
9. Gerstmayr, J.: Exudyn Videos, YouTube. https://www.youtube.com/playlist?list=PLZduTa9mdcmOh5KVUqatD9GzVg_jtl6fx (2023). Accessed 8 Mar 2023
10. Gerstmayr, J., Irschik, H.: On the correct representation of bending and axial deformation in the absolute nodal coordinate formulation with an elastic line approach. J. Sound Vib. **318**(3), 461–487 (2008)
11. Gonçalves, J., Ambrósio, J.: Advanced modelling of flexible multibody systems using virtual bodies. Comput. Assist. Mech. Eng. Sci. **9**, 373–390 (2002)
12. González, F., Luaces, A., Lugrís, U., González, M.: Non-intrusive parallelization of multibody system dynamic simulations. Comput. Mech. **44**, 493–504 (2009)
13. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010). Accessed 8 Mar 2023
14. Holzinger, S., Gerstmayr, J.: Time integration of rigid bodies modelled with three rotation parameters. Multibody Syst. Dyn. **53**(4), 345–378 (2021)
15. Holzinger, S., Schieferle, M., Gerstmayr, J., Hofer, M., Gutmann, C.: Modeling and parameter identification for a flexible rotor with impacts. J. Comput. Nonlinear Dyn. **17**(5), 051008 (2022)
16. Jakob, W., Rhinelander, J., Moldovan, D.: pybind11 – seamless operability between C++11 and Python. https://github.com/pybind/pybind11 (2016). Accessed 8 Mar 2023
17. Lam, S.K., Pitrou, A., Seibert, S.: Numba: a LLVM-based Python JIT compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM'15, New York, NY, USA (2015)
18. Mantegazza, P., et al.: Package mbdyn (Python), Sourceforge. https://mbdynsimsuite.sourceforge.net/project_doc/api/html/mbdyn-module.html (2007). Accessed 8 Mar 2023
19. Masarati, P., Morandini, M., Quaranta, G., Mantegazza, P.: Open-source multibody analysis software. In: Ambrósio, J.A.C. (ed.) Proceedings of Multibody Dynamics 2003, IDMEC/IST, Lisbon, Portugal (2003)
20. Masarati, P., Morandini, M., Mantegazza, P.: An efficient formulation for general-purpose multibody/multiphysics analysis. J. Comput. Nonlinear Dyn. **9**(4), 041001 (2014)
21. Mazhar, H., Heyn, T., Pazouki, A., Melanz, D., Seidl, A., Bartholomew, A., Tasora, A., Negrut, D.: CHRONO: a parallel multi-physics library for rigid-body, flexible-body, and fluid dynamics. Mech. Sci. **4**, 49–64, 02 (2013)
22. Negrut, D., Serban, R., Mazhar, H., Heyn, T.: Parallel computing in multibody system dynamics: why, when, and how. J. Comput. Nonlinear Dyn. **9**(4), 041007 (2014)
23. OpenAI: ChatGPT, OpenAI Inc./OpenAI LP. https://chat.openai.com (2023). Accessed 8 Mar 2023
24. Pechstein, A., Gerstmayr, J.: A Lagrange-Eulerian formulation of an axially moving beam based on the absolute nodal coordinate formulation. Multibody Syst. Dyn. **30**(3), 343–358 (2013)
25. Pieber, M., Gerstmayr, J.: Six-bar linkages with compliant mechanisms for an adaptive robot. In: International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, 44th Mechanisms and Robotics Conference (MR), vol. 10, DETC2020-22546 (2020)
26. Pieber, M., Neurauter, R., Gerstmayr, J.: An adaptive robot for building in-plane programmable structures. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 1–9 (2018)
27. Pieber, M., Ntarladima, K., Gerstmayr, J.: A hybrid arbitrary Lagrangian Eulerian formulation for the investigation of the stability of pipes conveying fluid and axially moving beams. J. Comput. Nonlinear Dyn. **17**(5), 051006 (13 pages) (2022)

28. Schöberl, J.: NETGEN an advancing front 2D/3D-mesh generator based on abstract rules. Comput. Vis. Sci. **1**, 41–52 (1997)
29. Schöberl, J.: C++11 implementation of finite elements in NGSolve. ASC Report 30/2014. https://www.asc.tuwien.ac.at/~schoeberl/wiki/publications/ngs-cpp11.pdf (2014). Accessed 8 Mar 2023
30. Tasora, A., Serban, R., Mazhar, H., Pazouki, A., Melanz, D., Fleischmann, J., Taylor, M., Sugiyama, H., Chrono, D.N.: An open source multi-physics dynamics engine. In: Kozubek, T., Blaheta, R., Šístek, J., Rozložník, M., Čermák, M. (eds.) High Performance Computing in Science and Engineering, pp. 19–49. Springer, Cham (2016)
31. Todorov, E., Erez, T., Tassa, Y.: Mujoco: a physics engine for model-based control. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 5026–5033. IEEE (2012)
32. Zhang, Q., Xu, L., Zhang, X., Xu, B.: Quantifying the interpretation overhead of Python. Sci. Comput. Program. **215**, 102759 (2022)
33. Zwölfer, A., Gerstmayr, J.: The nodal-based floating frame of reference formulation with modal reduction. Acta Mech. **232**(3), 835–851 (2021)